THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

GENERATION AND ORGANIZATION OF BEHAVIORS FOR AUTONOMOUS ROBOTS

JIMMY PETTERSSON

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2006

Generation and organization of behaviors for autonomous robots

JIMMY PETTERSSON
ISBN 91-7291-833-0

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2515
ISSN 0346-718X

Department of Applied Mechanics
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 1000

Chalmers Reproservice
Göteborg, Sweden 2006

Till Anita och Nils-Åke

Generation and organization of behaviors for autonomous robots

JIMMY PETTERSSON
Department of Applied Mechanics
Chalmers University of Technology

Abstract

In this thesis, the generation and organization of behaviors for autonomous robots is studied within the framework of behavior-based robotics (BBR). Several different behavioral architectures have been considered in applications involving both bipedal and wheeled robots. In the case of bipedal robots, generalized finite-state machines (GFSMs) were used for generating a smooth gait for a (simulated) five-link bipedal model, constrained to move in the sagittal plane. In addition, robust balancing was achieved, even in the presence of perturbations. Furthermore, in simulations of a three-dimensional bipedal robot, gaits were generated using clusters of central pattern generators (CPGs) connected via a feedback network. A third architecture, namely a recurrent neural network (RNN), was used for generating several behaviors in a simulated, one-legged hopping robot. In all cases, evolutionary algorithms (EAs) were used for optimizing the behaviors.

The important problem of behavioral organization has been studied using the utility function (UF) method, in which behavior selection is obtained through evolutionary optimization of utility functions that provide a common currency for the comparison of behaviors. In general, the UF method requires the use of simulations. Thus, an important part of this thesis has been the development of a general-purpose software library (UFLib) implementing the UF method. In order to study the properties of the UF method, several behavioral organization problems, mostly involving wheeled robots, have been considered. Most importantly, it was found that the UF method greatly simplifies the search for solutions to a wide variety of behavioral organization problems and requires a minimum of hand-coding. Furthermore, the results show that the use of multiple simulations (for the evaluation of a robot) significantly improves the ability of the robot to select appropriate behaviors. For the EA, it was found that the standard crossover procedure, which swaps entire utility functions between individuals, performed at least as well as several modified operators, and that the mutation rate should be set so as to generate around three parameter modifications per individual. Finally, some early results are presented concerning the use of the UF method in connection with a robot intended for transportation and delivery.

Keywords: autonomous robots, behavioral organization, action selection, utility function method, evolutionary robotics.

List of publications

This thesis is based on the work contained in the following papers, referred to by Roman numerals in the text:

I. Pettersson, J., Sandholt, H., and Wahde, M., *A flexible evolutionary method for the generation and implementation of behaviors for humanoid robots*, in: Proceedings of the IEEE-RAS International Conference on Humanoid Robots, Humanoids 2001, Tokyo, Japan, November 2001, pp. 279–286.

II. Pettersson, J., *EvoDyn: A simulation library for behavior-based robotics*, Technical Report, Chalmers University of Technology, September 2003.

III. Pettersson, J. and Wahde, M., *Application of the utility function method for behavioral organization in a locomotion task*, IEEE Transactions on Evolutionary Computation, Volume 9, Issue 5, Oct. 2005, pp. 506–521.

IV. Wolff, K., Pettersson, J., Heralić, A., and Wahde, M., *Structural Evolution of Central Pattern Generators for Bipedal Walking in 3D Simulation*, to appear in: Proceedings of the 2006 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2006), Taipei, Taiwan, 2006.

V. Pettersson, J. and Wahde, M., *UFLibrary: A Simulation Library Implementing the Utility Function Method for Behavioral Organization in Autonomous Robots*, submitted to: International Journal on Artificial Intelligence Tools, 2005.

VI. Wahde, M., Pettersson, J., Sandholt, H., and Wolff, K., *Behavioral Selection using the Utility Function Method: A Case Study Involving a Simple Guard Robot*, in: Proceedings of the 3$^{\mathrm{rd}}$ International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005), Fukui, Japan, 2005, pp. 261–266.

VII. Pettersson, J., Sandberg, D., Wolff, K., and Wahde, M., *Behavioral selection in domestic assistance robots: A comparison of different methods for optimization of utility functions*, to appear in: Proceedings of the 2006 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2006), Taipei, Taiwan, 2006.

VIII. Pettersson, J. and Wahde, M., *Improving generalization in a behavioral selection problem using multiple simulations*, in: Proceedings of the Joint 3^{rd} International Conference on Soft Computing and Intelligent Systems and the 7^{th} International Symposium on advanced Intelligent Systems (SCIS & ISIS 2006), Tokyo, Japan, 2006, pp. 989–994.

IX. Wahde, M. and Pettersson, J., *A General-purpose Transportation Robot: An Outline of Work in Progress*, in: Proceedings of the 15^{th} IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 06), Hatfield, United Kingdom, 2006, pp. 722–726.

X. Pettersson, J. and Wahde, M., *Behavior selection for localization and navigation in a transportation robot, using evolvable internal state variables*, manuscript to be submitted to: Autonomous Robots.

In addition to the papers listed above, the author has also been involved in the work presented in references [57], [59], [69], [70], and [71].

Acknowledgments

I would like to thank Chalmers Center for Mechatronics and System Engineering (CHASE) and the Carl Trygger Foundation for financial support for my research project.

A special thanks goes to my thesis advisor Mattias Wahde for his never-ending enthusiasm and immense competence. Without his excellent guidance and supervision during these years, this thesis would never have been. Thank you.

I would also like to thank all colleagues (present and former) at the Department of Applied Mechanics, who have contributed to make Chalmers such an enjoyable working place. I am also grateful to my co-workers at Waseda University in Tokyo.

Finally, a very special thank you to Marie for your patience and love, and for being by my side throughout this journey.

Jimmy Pettersson
Göteborg, 2006

Technical terms used in the thesis

The technical terms used in this thesis are listed below. For each term, the page number for its first occurrence is also given.

Table of contents

Chapter 1

Introduction and motivation

The theme of this thesis is the generation and organization (selection) of behaviors for **autonomous robots**, and the main objectives are (1) to find suitable representations for robotic behaviors in different contexts (see Papers I, III, IV, and V), (2) to improve and expand the **utility function method** (UF method) for behavioral organization (see Papers III, V, VI, VII, and VIII), and (3) to extend the behavior-based approach (described in Chapter 2) to real-world applications (considered in Paper IX and Paper X).

Applications involving both bipedal robots and wheeled robots are considered. In the case of bipedal robots, the problem of generating motor behaviors for walking and balancing is studied (Papers I, III, and IV) whereas, for the case of wheeled robots, both behavior generation and **behavioral organization** are considered (Papers VI–X). In Paper IV, the problem of behavioral organization is also considered for the case of a one-legged hopping robot.

For the organization of behaviors, the recently developed utility function (UF) method [66] is investigated by means of computer simulations, which have been focused on the study of the basic properties of the method. An important part of the work presented here, described in detail in Chapter 7 and in Papers V–VIII, is the development of a general-purpose simulation library (UFLIB), implementing the UF method.

Thus, the work in thesis has been carried out within the framework of **behavior-based robotics** (BBR) [3], in which robots are commonly provided with a repertoire of behaviors and a behavior selection mechanism for selecting the most appropriate behavior at all times. In BBR, the **behavioral repertoire** normally consists of a set of basic behaviors that, when combined by the behavioral organizer, gives rise to a more complex, overall behavior of the robot.

Motor behaviors handle the control of the actuators of a robot and represent a

special case included in the more general concept of behaviors in BBR. Examples of such general behaviors include: *find energy source*, *avoid obstacles*, and *follow wall*.

The emphasis of this thesis is on the use of biologically inspired computation methods, and, in particular, the use of **evolutionary algorithms** (EAs) [8, 27] in connection with (1) behavior generation based on several different architectures (see Chapters 2 and 3), and (2) the generation of behavior selection systems (see Chapters 5 and 6). EAs is the umbrella term for a class of powerful search methods based on the principles of biological evolution [8]. Such methods have been developed over the last few decades and are applicable to a large variety of optimization problems and, in particular, problems involving large and complex search spaces such as the behavioral organization problems studied here. Furthermore, the choice of biologically inspired computation methods is motivated by the biological roots of the behavior-based paradigm and, in the particular case of EAs, the remarkable innovation and adaptation resulting from natural evolution [12, 13].

Most of the work reported in this thesis has been carried out in simulations, a near-necessity when using evolutionary methods, since the alternative procedure of evaluating a robotic brain[1] in hardware is both much more time-consuming and requires continuous monitoring. However, some tests are underway where results from simulations are being transferred to a real robot. In addition, the author is involved in a project aimed at constructing a general-purpose transportation robot (see Papers IX and X), in which the UF method will be used for generating the behavior selection system.

Recent development of hardware and the increased availability of computational power have contributed to the rapid development of autonomous robots, i.e. freely moving robots. In contrast to manipulators (industrial robots), which are normally operating in structured environments, autonomous robots are designed to operate in **unstructured environments**. Hence, researchers in the field of autonomous robotics face the problem of developing robots capable of functioning in environments full of uncertainties. In such unpredictable environments it is difficult to define, *a priori*, a control strategy that is sufficiently general for the robot to perform its task(s) reliably. Traditional adaptive control [58] does, of course, allow parameters in the control system to change, but the alterations generated by such modifications do not amount to a complete *switch* from, for instance, a *walking* behavior to a *retrieve object* behavior. Hence, even though the output from classical controllers varies in response to deviations from the stipulated reference trajectories, fundamental changes in the behavior of such a controller do

[1] Henceforth, the term *robotic brain* will be used to denote the system that controls a robot.

not occur (although methods such as gain scheduling do allow a certain degree of flexibility). For this to happen, several distinct behaviors are needed and, equally important, a method for selecting the appropriate behavior at any instant.

The methods used should thus be such as to allow the robot to make its own decisions in the best possible way, using as input the sensory information gathered as it moves through the environment. In the case of bipedal robots, most work to date [18, 19, 71] has been focused on providing the robots with a reliable gait using classical control theory. The use of methods from classical control theory, such as PID controllers, is well motivated in cases where reference trajectories are readily available, but in the context of unpredictable environments, more adaptive methods, capable of adapting to changes in the environment, are required. One such architecture was used in Paper IV to generate bipedal gaits, where a full 3D model of a bipedal robot displayed a stable walking pattern based on interconnected central pattern generators (see Section 2.1.3).

The use of reference trajectories is still important for the generation of low-level behaviors such as local motor control [1]. For instance, keeping a bipedal robot in a certain posture can be achieved by means of reference points. However, when the bipedal robot suffers a strong perturbation in such a way that it loses its balance, it is advantageous to have the possibility quickly to switch to a different (rescue) behavior, in order for the robot to maintain its posture. Hence, methods that do not rely on pre-defined reference trajectories, such as biologically inspired methods, become a suitable choice (see also [11, 42]). It should also be mentioned that, even though classical control is commonly used in connection with robotic arms (manipulators), biologically inspired methods *can* be used in such cases as well, as shown in [46] where obstacle avoidance, without the use of reference trajectories, was evolved for a robotic arm.

Wheeled robots, which are considered in Papers VI–X, rarely require active balancing since they are stable by their design. Nevertheless, wheeled robots face difficult problems of behavior selection, just as any robot does regardless of its morphology. For example, consider a robot whose task it is to transport objects from its current location to a given target location (see also [14, 62, 64] and Papers IX–X). As the robot travels towards the goal, the behavior selection system has to take into account a number of different factors in order to activate the most appropriate behavior at all times. For instance, if the path of the robot contains many obstacles, should the robot wait or take an alternative path towards the goal? Should the battery be re-charged before traveling further? How far is it to the nearest charging station? How far away is the goal? Is the current estimate of the position accurate or does the odometry require calibration? Thus, the robot is faced with a behavior selection problem that inevitably involves trade-offs. A large part of this thesis is devoted to the solution of such problems, using

a behavior-based approach and evolutionary optimization of utility functions (see Chapter 6).

1.1 Contributions

The main contributions in this thesis is the improvement and expansion of the UF method for behavior selection in autonomous robots. Specific improvements include the introduction of behavioral hierarchies and customized evolutionary operators as well as, more recently, the addition of evolvable internal abstract (hormone) variables. The UF method is, to the author's knowledge, the only published general-purpose method for behavior selection in autonomous robots that delegates the generation of the behavior selection system to an automated process (an evolutionary algorithm, in the case of the UF method), thus minimizing the amount of manual parameter-tuning. Another important contribution is the development of the UFLIB simulation library, which provides a complete and user-friendly implementation of the UF method. With this library, a user can rapidly generate an executable file suitable for the problem at hand, particularly in cases where off-the-shelf behaviors can be used.

In addition, contributions have been made to the sub-field of gait generation for legged robots. Gaits have been developed and optimized (e.g. with respect to energy usage) using different representations, such as finite-state machines and central patterns generators (as well as other types of recurrent neural networks). A simulation library (EVODYN) for evolutionary optimization of tree-structured multi-body systems has also been developed.

The author of this thesis has been the sole contributor to Paper II, and the main contributor to Papers I, III, VII, and VIII. In Papers V, VI, IX, and X the authors have made approximately equal contributions. In Paper IV, the author's contribution was more limited.

Chapter 2

Behavior-based robotics

The field of behavior-based robotics (BBR) is mainly concerned with autonomous robots, which are supposed to function in unstructured environments, where robots often find themselves in situations they have not previously encountered [3, 7]. In BBR, there is a strong coupling between perception and reaction, where sensors provide the main source of information. Unlike classical artificial intelligence (AI), explicit internal world models, in which planning processes operate, are not used, since they lead to very slow decision-making [45]. It should be noted that the research field is still very young, around 20 years [7], and is constantly developing. Up until now, behavior-based robots have been quite simple, and the current level of intelligence of such robots can be considered to be at the level of insects [45].

BBR makes use of a bottom-up approach for the generation of robotic brains, in which complex overall behavior emerges from the combination of several, simpler behaviors[1]. Examples of such simple behaviors are; *locate target*, *avoid obstacles*, *go forward*, *find energy source*, and *stop*. Such simple behaviors could for instance, if organized in a suitable manner, provide a robot with the ability to navigate safely in a complex environment. As a specific example, in Paper VI, such a set of behaviors was used in an application involving a simple guard robot.

One of the main challenges in BBR, also treated in this thesis, concerns the organization of behaviors, that is, how the selection between different behaviors should be made at any given instant. Given information from relatively simple sensors, choosing the most appropriate behavior is often extremely difficult. Many methods rely on hand-coded settings [5, 37, 55]. While useful in simple cases, methods relying on hand-coded structures often lead to solutions that are not robust in more complex cases involving several behaviors [36]. In such cases, ethol-

[1]Note that behaviors are typeset in *italics*.

ogy provides an excellent source of inspiration [66], since even the simplest of animals are able to make intelligent choices (i.e. to carry out behavior selection), at least within their natural environments (see e.g. [10]). For a robot, an apparently simple task such as traveling from point A to point B in e.g. an office delivery application, may involve a multitude of different behaviors. For example, assuming that the robot has a map of the environment, either generated by the robot itself or defined by a user, a path can be generated so that it leads the robot from the current position (A) to its goal at B. A method for generating paths is described in Section 7.3.2 and in Paper X. Using the generated path, the robot first needs to activate its *navigation* behavior, in which the generated path towards B is followed. If obstacles are detected, the robot should activate its *obstacle avoidance* behavior in order to avoid collisions. After clearing any obstacles, the navigation towards B should be resumed. Assuming that the robot lacks a device for global positioning, it must update, continuously, the estimate of its current position, e.g. by means of odometry. As the robot travels towards B, drift in the odometry will cause the error between the estimated position and the robot's true position to increase.

In order to minimize the accumulated error in the odometry, a *localization* behavior must be activated, in which the robot uses its sensors, e.g. a camera or a laser scanning device, in combination with the map to identify its current position and calibrate the odometry. If the distance between A and B is relatively large, the localization behavior must be activated many times to maintain accuracy; *how many* is for the behavior selection mechanism to decide.

Problems like these, among others, are currently being investigated in a project involving a transportation robot, described further in Paper IX and Paper X. The general topic of behavioral organization is considered in Chapter 5 below. By contrast, in this chapter, the emphasis will be on the different architectures used for individual behaviors. Behaviors can be implemented in many different ways, and in the remainder of this chapter a few different behavioral architectures will be described. A discussion of different approaches to the generation of behaviors is given in Chapter 4.

2.1 Architectures for behaviors

Motor behaviors, i.e. behaviors for moving a robot or parts thereof, are the basic building blocks for generating the reactive behaviors typically used in behavior-based robots. As mentioned above and in Chapter 1, autonomous robots intended for unstructured environments require the use of adaptive techniques. Examples of architectures useful in the implementation of such systems are **if-then-else-rules** (see Paper I), **artificial neural networks** (ANNs) (see Paper III and [76]), and

Algorithm 2.1 Locate beacon behavior

```
1: if s > 0 then
2:     BeaconFound ← TRUE
3: else
4:     BeaconFound ← FALSE
5: end if
6: if BeaconFound then
7:     if s < ε then                    ▷ Stop when close to the beacon
8:         τ_L ← 0
9:         τ_R ← 0
10:    else                             ▷ Approach the beacon
11:        τ_L ← T
12:        τ_R ← T
13:    end if
14: else                                ▷ Turn
15:    τ_L ← −T
16:    τ_R ← T
17: end if
```

central pattern generators (CPGs) (see Paper IV), the latter being a special case of an ANN. These architectures share the advantage of being highly applicable for use in conjunction with biologically inspired computation methods, such as EAs, which are extensively used throughout this thesis. These three architectures will now be described in some detail.

2.1.1 If-then-else-rules

One of the quickest ways to generate simple behaviors is to use *if-then-else* constructs, where the experimenter implements the behavior directly in source code. By combining several rules, in a nested or sequential manner, it is usually fairly straightforward to implement simple behaviors. If a behavior tends to become complex, one should try to divide the behavior into several, less complex sub-behaviors, all in the spirit of behavior-based robotics. As an example, in Paper VI, the behavior *energy maintenance* was divided into the two behaviors *corner seeking*[2] and *battery charging*. An equivalent of the *corner seeking* behavior is shown in Algorithm 2.1, in which a differentially steered robot is supposed to locate an infrared (IR) beacon in the environment. A typical path, partly generated by this behavior, is shown in Fig. 2 of Paper VI. If active, the behavior shown in Algo-

[2]Charging stations were located in the corners of the arena.

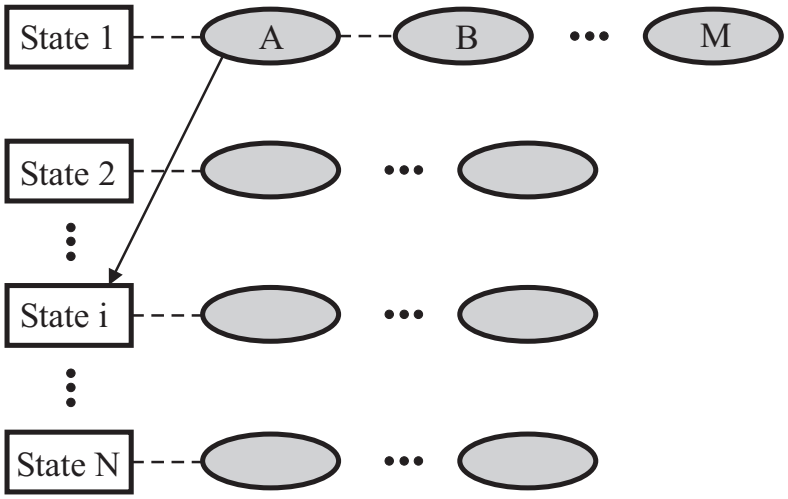


Figure 2.1: A generalized finite state machine with N states. For each state, there is a number of conditional transitions (the conditions are denoted A, B, ..., M for the first state). The arrow indicates a conditional transition (from state 1 to state i) that occurs when condition A in state 1 is satisfied. Since the conditions are checked in sequence (from left to right), condition A has higher priority than condition B.

rithm 2.1 will first query the beacon detector reading (s). This detector is a sensor placed on the robot's exterior, and its reading is proportional to the distance to the beacon (if detected, given the rather narrow (four degrees) field-of-view). If the sensor cannot detect any beacon ($s = 0$), the robot will start to turn by setting the motor commands (τ_L and τ_R) to values equal in magnitude but having opposite signs. In case a beacon is detected ($s > 0$), the robot will approach the beacon in an asymptotically straight line by setting the motor commands to equal, positive values. If the detector is positioned sufficiently close to the beacon ($s < \varepsilon$), the robot stops. Here, ε is a parameter to be tuned by the experimenter. *if-then-else* constructs were used in the implementation of all behaviors in Papers V–VIII, and X.

Generalized finite state machines (GFSMs) Generalized finite state machines [72], which are a special case of *if-then-else*-rules, consist of a finite number of states, as shown in Fig. 2.1. In GFSMs, each state is associated with a specific variable setting, whereas in ordinary finite state machines (FSMs) it is the state transitions that are associated with the actions taken [68]. Conditional transitions occur between the available states. If a condition is satisfied, the GFSM immediately jumps to the corresponding target state. If no condition is satisfied, no jump occurs and the GFSM remains in the same state. The conditions are checked se-

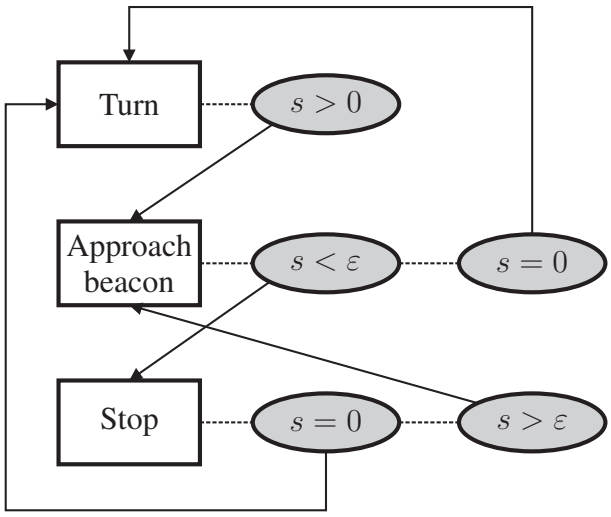


Figure 2.2: A generalized finite state machine (GFSM) implementing the behavior *locate beacon* for a differentially steered robot, also defined in Algorithm 2.1. The GFSM has three states, each associated with a motor command (Turn, Approach beacon, Stop) and a number of conditional transitions. Jumps between states are indicated by arrows. See the main text for the definition of s and ε.

quentially, going from left to right (see Fig. 2.1). Thus, the conditions associated with a particular state have different priorities. For example, in Fig. 2.1, condition A has higher priority than condition B which, in turn, has higher priority than condition M.

In Paper I, GFSMs were used as the architecture for bipedal motor behaviors. In that paper, an evolutionary algorithm (see Section 3.1) was used for the optimization of both the parameters and the structure of the GFSM. For the simulations in Paper I, a five-link bipedal robot, constrained to move in the **sagittal plane**, was used. As initial condition to the evolutionary algorithm, a rough indication of the desired behavior was specified (cf. initial reference values). The EA then performed further optimization of the behaviors. The method was applied to two test cases, namely (1) bipedal walking on a horizontal surface while using minimal amounts of energy and (2) robust balancing in the presence of impulsive perturbations.

Because of the modular nature of GFSMs, the architecture is very useful in the context of behavior-based robotics. It has also been shown that GFSMs allow for the combination of simple behaviors to form a more complex, composite behavior [72]. As a further example, a GFSM representation of the *locate beacon* behavior shown in Algorithm 2.1 is presented in Fig. 2.2.

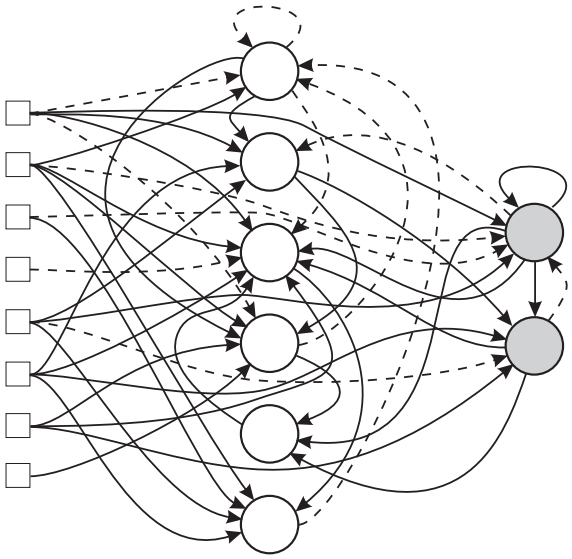


Figure 2.3: An example of a recurrent neural network with eight input elements (shown as boxes) and two output neurons (filled circles). Solid lines indicate positive (excitatory) weights, whereas dashed lines indicate negative (inhibitory) weights (see also Fig. 7 in Paper III).

2.1.2 Recurrent neural networks (RNNs)

An RNN is an artificial neural network [25] consisting of a number of neurons (nodes) with arbitrary connections (including self-coupling of individual neurons). An example of such a network is shown in Fig. 2.3. RNNs can operate either in discrete time, as is common in **feedforward networks** (i.e. ANNs without feedback connections), or in continuous time. In the latter case, using a simple neuron model, the dynamical behavior of the i^{th} node in the network is governed by the equation

$$\tau_i \dot{y}_i + y_i = \sigma\left(b_i + \sum_j w_{ij} y_j + \sum_j w_{ij}^I I_j\right), \quad i = 1, \ldots, n, \tag{2.1}$$

where n is the number of neurons in the network, τ_i are time constants, y_j is the output (activity) of node j, w_{ij} is the (synaptic) weight connecting node j to node i, w_{ij}^I is the weight connecting input node j to node i, I_j is the j^{th} external input to node i, and b_i is the bias term, which determines the output of the neuron in the absence of inputs. $\sigma(\cdot)$ is a sigmoidal function whose main purpose is to restrict the activity of the neurons to a given range. A common choice for the sigmoidal function is $\sigma(z) = \tanh(cz)$, where c is a constant, in which case the neuron output is restricted to the range $[-1, 1]$.

Albeit with strong simplifications, RNNs are based on biological neural networks, in which recurrent couplings are almost always present [30]. In [57] a recurrent neural network for balancing a robotic leg was generated using a simple EA. In Paper III, a more complex EA allowing structural modifications was applied to the same problem, using a more sophisticated dynamical simulation of the balancing leg (see Paper II). ANNs (including RNNs) possess a number of important properties such as graceful degradation in the case of neuron loss, and the ability to generalize to situations not previously encountered, as discussed in Paper III and [39].

2.1.3 Central pattern generators (CPGs)

In experiments involving animals [23], e.g. cats, oscillatory motor patterns have been found in cases where sensory feedback had been removed (by decoupling of the central nervous system). The neural networks found to be responsible for generating these rhythmic phenomena have been termed central pattern generators (CPGs). Each pattern generator is a cluster of interconnected neurons that operate in an autonomous manner, i.e. a CPG is capable of producing oscillatory output without any rhythmic input.

For obvious reasons, in the case of humans, there are fewer experimental results concerning the neural control of walking. However, measurements from motor neurons taken during normal walking suggest that CPGs play a central role also in humans [28, 47, 77] even though the higher complexity of bipedal walking has resulted in a more complex interaction between the motor cortex and the spinal cord than in quadrupeds.

Inspired by the above findings in neuroscience, models of CPGs have been proposed for use in artificial systems such as e.g. robots. A further motivation for the use of such models is their ability to operate without reference trajectories. Three main models have been proposed in the literature: (1) the closed-loop model, (2) the pacemaker model, and (3) the half-center model [38]. Here, only the half-center model will be described. For a description of the closed-loop model and the pacemaker model, see [60]. The half-center model, shown in Fig. 2.4, was used in Paper IV, in which a bipedal gait was generated in a 3D simulation of a bipedal robot having 14 **degrees of freedom** (DOFs). Trajectories for a few of the CPG-driven joints in the bipedal robot are shown in Fig. 2.5.

As described in Paper IV, the mathematical model of the half-center CPG,

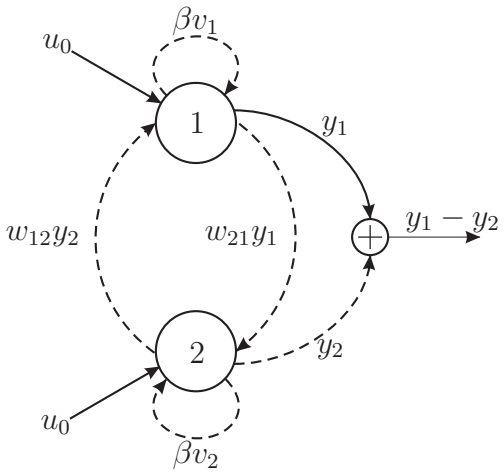


Figure 2.4: A half-center central pattern generator. Dashed lines indicate inhibitory connections whereas solid lines indicate excitatory connections.

shown in Fig. 2.4, can be summarized by the following set of equations:

$$\tau_u \dot{u}_i + u_i = u_0 + \beta v_i + \sum_{j=1}^{n} w_{ij} y_j, \tag{2.2}$$

$$\tau_v \dot{v}_i + v_i = y_i, \tag{2.3}$$

$$y_i = \max(0, u_i), \tag{2.4}$$

where τ_u and τ_v are time constants, u_0 is an external, non-oscillating input, w_{ij} is the weight of the connection going from neuron j to neuron i, y_i is the output of neuron i, and n is the number of neurons. u_i and v_i are the state and self-inhibition of neuron i, respectively. β is a parameter used for modulating v_i. By changing the value of the external (non-oscillating) input u_0, the amplitude of the output from the CPG can be varied. The frequency of the oscillatory output can be changed by adjusting the time constants τ_u and τ_v.

CPGs, such as the one shown in Fig. 2.4, are commonly used in connection with a feedback network. In Paper IV the feedback network, linking the CPGs to form a robotic brain generating a gait, was optimized by means of an EA.

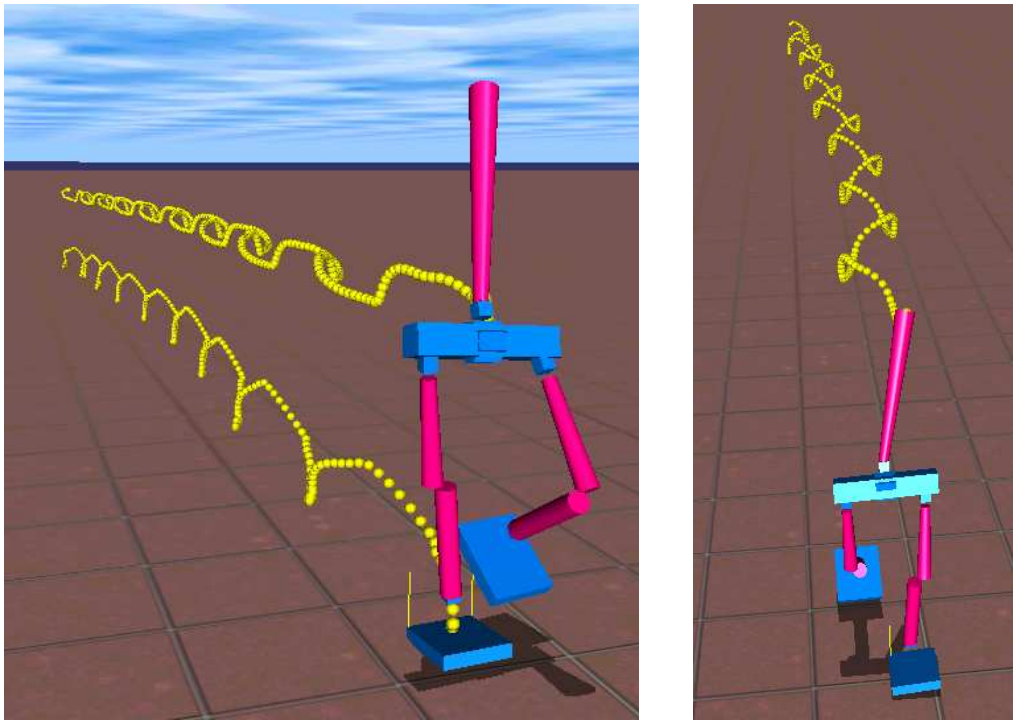


Figure 2.5: Bipedal gait generated by evolutionary optimization of central pattern generators in Paper IV. The small spheres indicate the generated trajectories of two of the 14 available joints (left panel) and the tip of the upper body (right panel).

Chapter 3

Evolutionary robotics

Evolutionary robotics (ER) is a subfield of robotics in which evolutionary algorithms (EAs) are used in the construction[1] of robots [67, 48]. Here EAs are considered as being the umbrella term for the entire family of evolutionary methods. A general introduction to EAs will be given in Section 3.1 and in Section 3.2 different ways to represent individuals (i.e. the candidate solutions found by the EA) will be described.

Since EAs often require many evaluations before a satisfactory solution is found, ER normally relies on simulations, a topic that will be considered in Section 3.3.1 below. However, even when simulations are used, the ultimate aim is, of course, to implement the results in real robots. An alternative to using simulations is to run the evolutionary procedure directly in hardware. Such approaches are briefly considered in Section 3.3.2.

It should be noted that EAs can be used in the optimization of individual behaviors as well as in the optimization of behavior selection mechanisms. The latter is the topic of Chapters 6 and 7, where behavior selection is obtained by means of evolutionary optimization of utility functions.

3.1 Evolutionary algorithms

EAs is the common term used for a class of search algorithms inspired by biological evolution. Examples of such algorithms are: **genetic algorithms** (GAs) [27], **genetic programming** (GP) [34], and **evolution strategies** (ES) [53].

EAs, the flow of which is shown in Algorithm 3.1, operate on a **population**, i.e. a group of individuals. A member of the population is referred to as an **indi-**

[1] The construction may involve the robot's brain, body, or both.

Algorithm 3.1 A typical evolutionary algorithm.

```
1: INITIALIZE population with random candidate solutions
2: EVALUATE candidates
3: repeat
4:     repeat
5:         SELECT parents based on fitness
6:         RECOMBINE pairs of parents through crossover
7:         MUTATE new candidates
8:     until NEW POPULATION formed
9:     REPLACE parents
10:    EVALUATE new candidates
11: until TERMINATION CRITERIA fulfilled
```

vidual (or candidate solution). The information needed to generate an individual is stored in its **genome**, which consists of one or several **chromosomes** that, in turn, contain a set of **genes** that encode information for the construction of an individual. The entire genome, with all its settings, is referred to as the **genotype**. In certain cases, as discussed in Section 3.2.2, the decoding step need not be applied, however. Through a process of decoding, the corresponding **phenotype**, i.e. the individual with all its traits, is constructed.

EAs are especially suited for problems involving large search spaces with many local optima (see [27], [44], and [8] as well as references therein for basic information regarding EAs); through the action of their various operators (see below), EAs carry out a non-local search, unlike gradient-based methods. In EAs, it is also possible to mix both continuous and discrete variables as long as a fitness (objective) function can be formulated. That is, it is not required that a gradient should be available in order for the EA to function properly. Another benefit is the possibility to use EAs in cases where it is difficult to derive an analytical model of the system, or when such a model simply does not exist [75]. EAs search through the space of possible solutions by applying genetic operators: **selection**, where individuals are chosen (for reproduction) stochastically in proportion to their fitness, **crossover**, responsible for exchanging genetic information between selected individuals, and **mutation**, which is a stochastic operator responsible for introducing new material into the population. Through the use of these operators, the EA modifies the solutions and improves their performance according to a user-defined fitness function.

It should be noted that, normally, it is not possible to identify whether the best solution found by an EA represents a local optimum or the global optimum. However, in robotics problems, it is often possible to judge, simply by inspecting

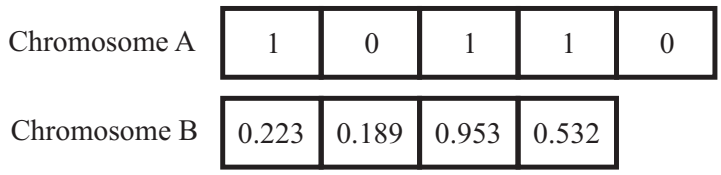


Figure 3.1: Example of chromosomes using binary encoding (A) and real-number encoding (B).

the robot in action, whether its evolved brain performs adequately.

3.2 Encoding schemes

There exists many different ways of encoding an individual, such as strings of digits (usually referred to as chromosomes in the case of GAs), arrays, lists, and trees. Chromosomal encoding is considered in Section 3.2.1. Another possibility, described in Section 3.2.2, is to implement an EA in such a way that the genetic operators (crossover and mutation) act directly on the target structure, i.e. without using the intermediate steps of encoding and decoding.

3.2.1 Chromosomal encoding

In chromosomal encoding, individuals are represented by strings of digits. The number of genes in the chromosome depends on the number of parameters needed in the problem at hand. Each individual will represent a point in the space spanned by these parameters.

Although many different encodings of chromosomes are possible, the most commonly used are **binary encoding** and **real-number encoding**, as exemplified in Fig. 3.1. These chromosomes generate, via a decoding process, an individual that represents a possible solution to the problem considered. The decoding procedure is decided upon by the experimenter and is problem-dependent. As an example, chromosome A in Fig. 3.1 may be decoded to a floating point value as

$$x_{\text{float}} = 1 \times 2^{-5} + 0 \times 2^{-4} + 1 \times 2^{-3} + 1 \times 2^{-2} + 0 \times 2^{-1} = 0.40625, \quad (3.1)$$

or to an integer value as

$$x_{\text{integer}} = 1 \times 2^{4} + 0 \times 2^{3} + 1 \times 2^{2} + 1 \times 2^{1} + 0 \times 2^{0} = 22, \quad (3.2)$$

depending on how the user chooses to implement the decoding procedure.

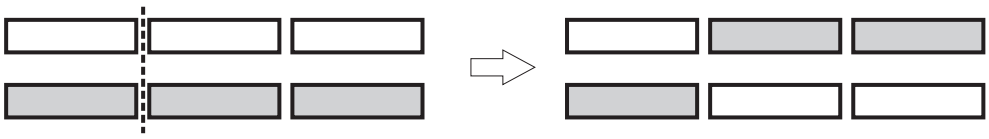


Figure 3.2: Single-point crossover using fixed-size chromosomes. The dashed line indicates the randomly chosen crossover point.

As an example of chromosomal encoding, consider the chromosomes used in UFLIB, the software library implementing the utility function method (see Chapters 6 and 7), in which behavior selection is achieved through evolutionary optimization of utility functions. Each utility function is generated using a polynomial ansatz of a given degree. For example, a second degree polynomial, having two dependent variables (s and p), is given by

$$U(s,p) = a_{00} + a_{10}s + a_{01}p + a_{11}sp + a_{20}s^2 + a_{02}p^2, \tag{3.3}$$

where a_{ij} are the coefficients to be determined by the EA. Functions like the one above can be encoded in a chromosome having six real-valued genes (in a predefined range), one for each coefficient a_{ij}. Chromosomal encoding of this kind was used in Papers III, V–VIII, and X for the evolution of the behavior selection mechanism.

One of the benefits with chromosomal encoding is that the genetic operators for crossover and mutation are easy to implement (assuming fixed-size chromosomes). In case the chromosomes vary in length, the crossover operator must be modified in order to account for that fact, by assuring that the crossover procedure generates valid results. As an example, single-point crossover using fixed-size chromosomes is illustrated in Fig. 3.2.

3.2.2 Explicit encoding

In **explicit encoding** there are no chromosomes. Instead, the EA operates directly on the structures used for representing the individuals, i.e. the evolutionary operators modify the phenotype directly. Hence, in explicit encoding, there is no genotype and the intermediate decoding step is not needed. Using an analogy from object-oriented programming, in the explicit encoding approach each individual is an instance of a class that implements methods for both crossover and mutation (which are problem-dependent).

Such encodings were used in Paper I and Paper III, where, in the first case, the EA was applied directly to a GFSM, and in the latter case to an RNN. Regardless of the architecture used, the genetic operators must of course be defined such that

they are able to operate on that architecture, i.e. such that they always generate valid structures. In most applications of EAs, the genetic operators act on architectures of fixed size, and the optimization thus consists of parameter tuning. For example, in [57], an EA was applied to RNNs of fixed size. However, fixed-sized architectures have an obvious limitation: If the size is chosen in a bad way, it cannot be modified during the optimization procedure which then only samples a small part of the space of possible structures. An inappropriate choice of network architecture, for example one using too few neurons or feedback connections, can make it even *theoretically* impossible to solve the problem at hand. EAs can, however, be modified to optimize not only the parameters, but also the *structure* of the network. For instance, when using artificial neural networks, it may be difficult to specify the number of neurons in advance. Thus, it is beneficial to let the EA optimize both the weights and the number of neurons in the network. In such problems, the structure of the systems being optimized is allowed to vary in complexity during the optimization procedure.

For some architectures the crossover operator in an EA (see Section 3.1) is particularly hard to apply in a useful manner. This is especially true in architectures that have a distributed nature of computation, such as e.g. RNNs [76]. In addition, when the EA operates directly on the structures that are being optimized, as in Papers I and III, it is difficult to get the benefits of a crossover operator associated with chromosomal encoding.

Evolution of RNNs was one of the topics of Paper III. Here, crossover was implemented in two different ways: **intraspecies crossover**, where only networks with the same size were allowed to exchange material, and **single-neuron crossover**, where a single neuron was extracted from each selected network and added to the other network after removal of inter-neuron connections (see Fig. 3.3). However, the effect of these operators often amount to **macromutations**, i.e. large random changes which generally are detrimental (see Paper III). Intraspecies crossover suffers from the same problems; even though it operates on networks containing the same number of neurons, crossing parts of networks in which the incoming weights to the neurons may be totally different, generally causes a degradation in performance [74].

3.3 Approaches to ER

3.3.1 Evolution in simulation

Since evolutionary methods normally require many, often lengthy, evaluations before a satisfactory solution is found, simulations are often the most realistic al-

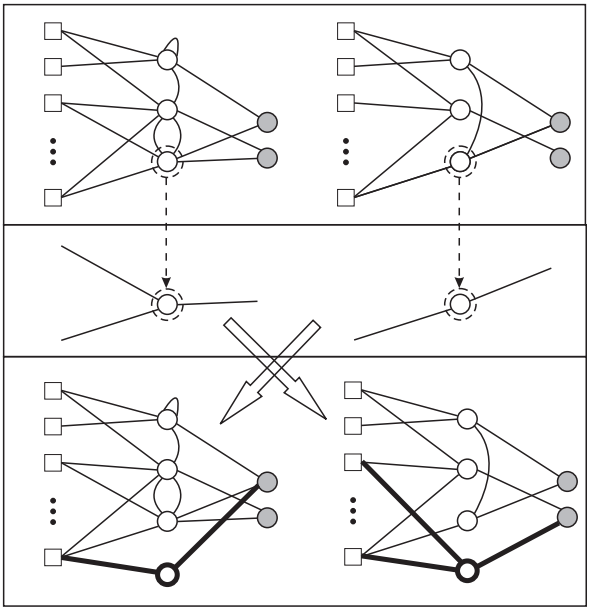


Figure 3.3: Example of single-neuron crossover between RNNs (see Paper III). The top part of the figure shows the original RNNs and the middle part displays the selected neurons with the inter-neuron connections removed. In the bottom part, the neurons are exchanged and inserted into the original RNNs, forming two new networks. Bold lines represent the added material.

ternative (as opposed to performing the evolution in real robots). For example, the investigation performed in Paper VII required runs in which 10^4 individuals were evaluated. Each of these runs took approximately 24 hours to complete on a standard PC equipped with a 3 GHz processor.

Performing realistic simulations is indeed a difficult task. No matter how much care is taken to make accurate models of the real world, there will still be a certain degree of discrepancy [29]. Individual components (e.g. sensors) used in a real robot may display considerable differences in performance. Furthermore, an idealized kinematic model can never represent a real robot exactly since, for example, the wheels of the robot will never be perfectly circular. While it is futile, in simulations, to attempt an exact representation of reality, a better approximation can be obtained by introducing noise in all relevant parts (e.g. sensors and actuators) of the simulations [29]. However, it should be noted that noise was *not* applied in many of the papers, e.g. most of the papers dealing with the UF method (Papers III, V–VIII). The aim of those papers was to study the basic properties of the UF method, a goal that was easier to reach using simulations that were deterministic (with the exception of the EA, which, of course, used stochas-

tic operators). The emphasis in those papers was placed on the *comparison* of, for example, the speed of evolution using different setups for the EA (see e.g. Table I in Paper VII). However, UFLIB (see Chapter 7) supports the inclusion of noise at all relevant levels and, in Paper X, which is aimed at generating a robot capable of solving a realistic transportation and delivery task, noise has been applied in all simulations.

The general flow of the robot simulations performed in the above-mentioned papers is illustrated in Fig. 3.4. This figure also serves as a general schematic for most simulations involving autonomous robots. As a first step, an initialization procedure is needed, in which the states of both the robot and the environment (arena) are set to some predefined initial state. For instance, an initial state could concern the initial position or the initial heading of the robot. For the environment, all moving obstacles must be placed in their respective starting positions. One should keep in mind that, if the performance of different robots is to be compared, which is the case when using an EA, it is important that each robot should be evaluated under the same conditions, i.e. that initial states remain unchanged between simulations. This is ensured in step 1 in Fig. 3.4. Once step 1 has been performed, steps 2–7 are iterated until a certain termination criterion is met (see the examples below). The size of the time step used for advancing the simulation must be carefully chosen; it needs to be sufficiently small to capture any modeled dynamic processes as well as to ensure stable numerical integration of the robot's equations of motion. In step 2, the simulation loop starts by updating the environment. Then, in step 3, the robot's sensors are updated, using the newly updated environment. Step 4, as implemented in UFLIB, consists of an update of the behavior selection system, in which the utility (see Chapter 6) associated with each behavior receives a new value. Based on these utility values, a behavior is selected for activation. In case the behavior is a motor behavior, the associated motor commands are then set accordingly in step 5. The motor commands are then passed to a motor model that produces torques (or forces) which, in turn, are used in step 6 to integrate the robot's equations of motion. Step 7 involves a check of the user-defined termination criteria (or criterion in case there is only one). Simulations are normally terminated if the maximum simulation time is reached, if the robot collides with another object, or if the battery energy level reaches zero. However, a user of UFLIB may add other termination criteria, or remove any of the existing ones by overriding the default implementation. In case no termination criterion is fulfilled, steps 2–7 are repeated.

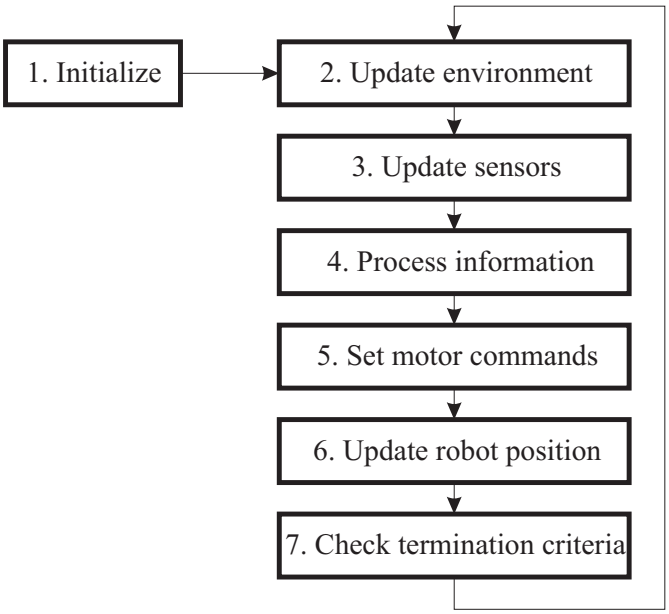


Figure 3.4: Schematic illustration of the flow in a typical simulation of an autonomous robot. Following initialization, steps 2–7 are carried out in each time step. Figure adapted from [68].

3.3.2 Evolution using hardware

Evolution using hardware, i.e. in real robots, can be carried out in several different ways. One common approach is to let the EA run on a separate computer, which uploads the robotic brain to a single real robot for evaluation [16]. The performance of the robot is then fed back to the EA for further processing. The procedure just described is an attractive alternative to the hand-tuning of parameters that is commonly needed when implementing results, obtained through simulations, in real robots. However, evolution using hardware is typically very time-consuming. For example, in the navigation experiments[2] investigated in [16], the evaluation (performed in hardware) took around 30 seconds per individual, compared to a small fraction of a second had the investigation been carried out in simulation. Furthermore, evolution using hardware generally requires continuous monitoring of the robots and, possibly, manual assignment of fitness values.

Another approach is called **embodied evolution** [15, 73], where the entire EA is run in real robots, i.e. each individual in the population is represented by a real robot. Exchange of genetic material is done by means of interfaces placed on

[2]The experiments involved straight-line navigation while avoiding obstacles as well as localization of a battery charger.

each robot, for example infrared communication ports. In this way, results can be achieved that are robust to the inevitable differences between supposedly identical hardware components.

An interesting hybrid approach is to first run an EA in simulation for, say, a few hundred generations, and then continue the run in a real robot for a few generations. Such an approach was used in [43] in connection with a simple navigation task. It was found that the fitness values dropped at the transition from simulations to evolution using hardware, but then increased quite rapidly.

In any case, despite their limitations, simulations of autonomous robots are important as an initial step towards the implementation in real robots.

Chapter 4

Behavior generation

There are several ways to generate behaviors for autonomous robots, e.g. hand-coding, evolutionary algorithms [48], simulated annealing [33, 35], particle swarm optimization [31, 52] etc. In this thesis, two approaches have been used, namely (1) hand-coding (Papers VI–VIII, and Paper X) and (2) EA-based optimization (Papers I, III, and IV).

For the particular case of motor behaviors, the implementation is strongly dependent on the morphology of the robot in which they are applied: Wheeled robots normally do not require active balancing, whereas statically unstable systems such as bipedal robots do. Thus, motor behaviors for wheeled robots are generally simpler and can therefore often be generated by hand (as described in Section 4.1), whereas motor behaviors for bipedal robots commonly require a more complex optimization procedure. Evolutionary optimization of GFSMs, RNNs, and CPGs (see Section 2.1) for the development of bipedal motor behaviors is considered in Section 4.2.

However, the use of evolutionary methods for the optimization of behaviors is in no way limited to bipedal robots; it is the architecture chosen to represent a behavior that guides the choice of optimization method. If, for instance, a *seek light* behavior in a wheeled robot is represented by an ANN, optimization by hand is unsuitable. Instead, an EA can be used for that purpose [16]. Alternatively, the *seek light* behavior might be simple enough to be implemented by hand using, for instance, *if-then-else*-rules as described in Section 2.1.1.

Algorithm 4.1 Obstacle avoidance behavior.

if $(s_1 > \varepsilon_R)$ or $(s_2 > \varepsilon_R)$ **then**
    $\tau_L \leftarrow -T$
    $\tau_R \leftarrow T$
else if $(s_4 > \varepsilon_L)$ or $(s_5 > \varepsilon_L)$ **then**
    $\tau_L \leftarrow T$
    $\tau_R \leftarrow -T$
else if $(s_2 > \varepsilon_C)$ or $(s_3 > \varepsilon_C)$ or $(s_4 > \varepsilon_C)$ **then**
    $\tau_L \leftarrow -T$
    $\tau_R \leftarrow -T$
else if $\bar{s} > \varepsilon_S$ **then**
    $\tau_L \leftarrow T$
    $\tau_R \leftarrow -T$
else
    $\tau_L \leftarrow 0$
    $\tau_R \leftarrow 0$
end if

4.1 Hand-coded behaviors for wheeled robots

Behaviors for wheeled robots[1], within the framework of UFLIB (see Chapter 7), are normally hand-coded using the *if-then-else*-rules described in Section 2.1.1. This is possible due to the fact that those behaviors normally have a straightforward logic in their operation and that they are simple enough to lend themselves to manual implementation in a differentially steered robot. For example, the logic for a simple realization, using IR sensors, of the commonly used behavior *obstacle avoidance* (B) (see Papers V–VIII) can be described as shown in Algorithm 4.1: When B is activated by the behavior selection mechanism, the motor commands are first set to zero, causing the robot to stop. If any of the two leftmost IR sensors (s_4 and s_5), shown in Fig. 4.1, have a non-zero reading larger than a threshold value ($\{s_4, s_5\} > \varepsilon_L$), the motor commands ($\tau_L$ and τ_R) are set in such a way that the robot rotates, in the clockwise direction and without moving its center of mass, until the detection disappears. An equivalent procedure is performed in the case of a detection by the two rightmost sensors, i.e. if $\{s_1, s_2\} > \varepsilon_R$. If, instead, any of the three central sensors have a reading larger than a certain threshold ($\{s_2, s_3, s_4\} > \varepsilon_C$), the motor commands are set so that the robot travels back-

[1] In this thesis, the wheeled robots considered are all differentially steered, i.e. equipped with two actuated wheels and a support, either in the form of a non-actuated wheel or a low-friction (e.g. teflon) sphere.

Algorithm 4.2 Straight-line navigation behavior.

1: $\tau_L \leftarrow T$
2: $\tau_R \leftarrow T$

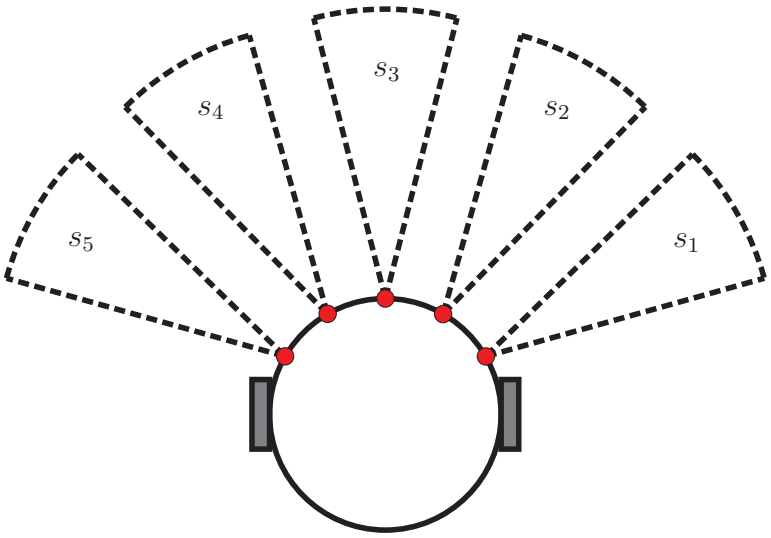


Figure 4.1: A differentially steered robot with five symmetrically placed infrared sensors.

wards, causing it to move away from the detected obstacle. In case a majority of the sensors have non-zero readings, i.e. the average of the sensor readings is larger than a certain threshold ($\bar{s} = 1/5 \sum_{i=0}^{5} s_i > \varepsilon_S$), the robot rotates without moving its center of mass until the readings disappear ($\bar{s} < \bar{s}_\varepsilon$).

Another simple example is the case of the *straight-line navigation* behavior (see Papers V–VIII), in which the motor commands are set to equal values, causing the robot to move forward in an asymptotically straight line. Note that, in the cases considered in Papers V–VIII, the robot did not need to use any sensor readings in this behavior, shown in Algorithm 4.2, since the selection of behaviors (e.g. the activation of the *obstacle avoidance* behavior described above) was handled by the behavior selection mechanism (see Chapter 6).

4.2 Evolved behaviors for bipedal robots

Three different representations for motor behaviors in bipedal robots have been considered in the papers included in this thesis, namely: (1) GFSMs, (2) RNNs, and (3) CPGs. These three representations have been applied to the problems of bipedal walking (Papers I and IV) and balancing in the presence of perturbations

(Paper III and [57]). In all cases, the architectures used explicit encoding, as described in Section 3.2.2, and were optimized by means of an EA (see Section 3.1) allowing both parameters and structures to change.

GFSMs, described in Section 2.1.1, were used in Paper I to produce an energy-optimized bipedal gait. In addition, a behavior for handling a sequence of impulsive perturbations was generated. In both cases the modular structure of the GFSM proved to be very useful and the best GFSM did indeed mimic the cyclic nature of human gait patterns, even though this was not explicitly enforced in any way. The EA was, however, initialized with a set of reference points that roughly specified a suitable trajectory. While not absolutely necessary, this was done in order to speed up the evolutionary process. In these energy optimization runs, the bipedal robot was given a fixed amount of energy and the distance walked was taken as the measure of fitness. Hence, energy efficient bipedal gait was generated implicitly. At the end of the simulations, the walking length had improved by 134%, using a GFSM consisting of only eight states (see Paper I).

Another approach was taken in Paper III, this time using the simulation library EvoDyn developed by the author (see Paper II) for the implementation of both the rigid body dynamics engine and the EA. As in [57], an RNN was chosen as the architecture for the motor behaviors to be optimized and a similar mechanical structure, consisting of a foot and a leg, was used. Both the foot and the leg were modeled as rigid bodies with distributed mass. The leg was actuated through two revolute joints and the entire system was free to move in space. Ground contact was modeled by two spring and damper models; one in the vertical direction and one in the horizontal direction. Vertical forces from those models served as representations of the pressure distribution under the foot (see Fig. 3 in Paper III).

In Paper IV, a bipedal gait was generated using central pattern generators (CPGs) as described in Section 2.1.3. CPGs, linked together via a feedback network were optimized using a GA, and were shown to generate a stable bipedal walking pattern.

Chapter 5

Behavioral organization

Behavioral organization, also known as action selection, behavioral coordination, or behavior selection, concerns the problem of how to select, at all times, an appropriate behavior for activation. Since the applicability of a given behavior often varies in a manner which is difficult to predict, the development of generally applicable systems for behavior selection is a daunting task. In BBR applications, robots are usually situated in dynamic environments and must be able to take the associated uncertainties into account. As mentioned in Chapter 2, in BBR, action selection occurs without the use of explicit world models of the kind normally employed in classical **artificial intelligence** (AI) [56]. The high-level, cognitive processes carried out in classical AI are often very slow [45]; in classical AI, a sense-model-plan-act sequence is executed [3], in which a model of the environment is first generated, and the robot then attempts to reason and plan within this model, trying various alternatives in order to find the appropriate action, which is then executed in the real world. Such an approach is evidently not very well suited to the case of robots operating in rapidly changing environments.

However, while robots in BBR often have a strong element of **reactivity** (i.e a direct coupling between perception and reaction) the use of, say, short-term memory and other non-reactive concepts is, of course, not excluded. For example, in the UF method (see Chapter 6), the internal abstract variables can provide the robots with a rudimentary short-term memory. The important distinction, compared to the case of classical AI, is that *explicit* modeling of the environment plays a much less important role in BBR. However, particularly for robots that are assigned complex tasks (such as the one considered in Paper X), a hybrid approach is often used in practice. Thus, for example, while the behavior-based robot considered in Paper X does make use of a behavior selection system and a set of basic behaviors with a high degree of reactivity, the robot is also equipped with a map

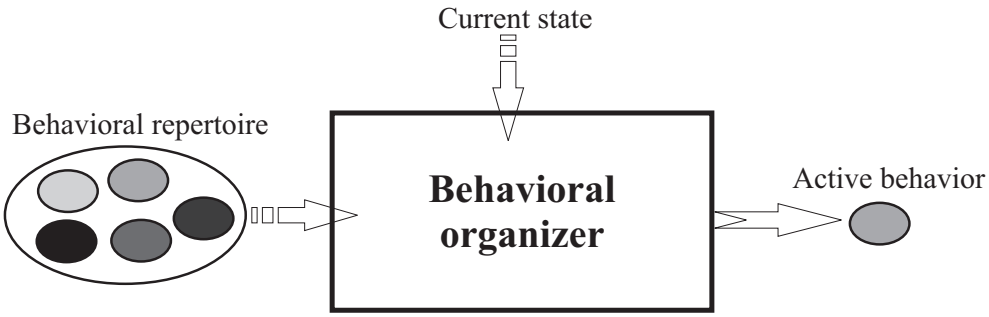


Figure 5.1: Schematic illustration, taken from Paper V, of a behavioral organizer for arbitration methods. The ellipses represent behaviors. Given the current state of the robot (defined by sensor readings and internal variables), a single behavior is selected from the repertoire of behaviors. The selected behavior is then given control of the robot until a different behavior is activated.

of the environment. It is important to note, however, that this map is used only for generating a navigation path upon activation of the *navigation* behavior. The robot does *not*, for example, traverse a set of virtual paths, between its current position and a goal position, in order to find the most suitable one; such an approach would not be very useful in cases where moving obstacles are present. Instead, in a BBR-like fashion (see [3], p. 191) the robot may, at any time, switch to a different behavior (e.g. *obstacle avoidance*) if the sensory information suggests that such a switch is needed, for example to avoid a collision.

Selecting the most appropriate behavior at any instant in time is one of the most important and most challenging problems in BBR. Many methods for behavioral organization have been suggested in the literature. A review of such methods and their taxonomy can be found in [49], where methods for behavioral organization are divided into **arbitration methods** in which a single behavior is selected for activation at any given time, and **command fusion methods** in which the action performed represents the weighted output of several behaviors. Arbitration methods include the **subsumption method** [7], in which a layered type of control is used, and **activation networks** where selection is performed according to activation levels based on exchanges of a quantity called activation energy between different parts of a network [37]. Examples of command fusion methods are the **potential field** method [32, 59], in which goals and obstacles are represented by artificial potential fields and action selection is based on the gradient of the combined fields, and **fuzzy command fusion** [50], in which fuzzy inferencing methods are used for producing a weighted output. A central part of this thesis concerns the use of the recently introduced utility function (UF) method [66] which is studied in Paper III, Papers VI–VIII, and Paper X, and is further

described in Chapter 6. An important aim of this method is to minimize, from a user's point of view, the amount of hand-tuning of parameters, a common feature in most current methods for behavior selection. In the UF method, simulations are conducted in which an evolutionary algorithm is used for the optimization of the behavioral organizer, whose role is illustrated in Fig. 5.1. As mentioned above, the purpose of the behavioral organizer is to select the most relevant behavior, in the case of arbitration methods, or to generate a weighted action, in the case of command fusion methods, based on the current state of the robot and its environment. Thus, the function of the behavioral organizer is commonly based on both available sensor data *and* the internal state of the robot.

As the aim in behavioral organization is to form complex robotic brains through the combination of several behaviors, i.e. to generate a complex *overall* behavior of the robot, there are also other approaches that do not involve a behavior selection mechanism. Within the framework of complex behaviors, several studies have been made involving the generation of complex behavior without the explicit use of a behavioral organizer. For instance, in [22], a procedure called **incremental evolution** was used, in which complex behaviors are evolved by making the robot's task progressively more difficult. Another approach is to divide a complex behavior into simpler sub-behaviors and evolve those separately. Once the behaviors have been generated, they can be fused, by means of an evolutionary method, into a composite behavior. This procedure was used in [72] to evolve a cleaning robot that was able also to avoid obstacles. Although complex overall behaviors can be achieved without the explicit use of behavioral organizers, methods such as those just mentioned, do not allow for the same degree of modularity found in e.g. arbitration methods, in which individual behaviors can be added to the behavioral repertoire [66] without regard to the architecture in which they were generated. In the next chapter, the UF method will be introduced and described in some detail.

Chapter 6

The utility function method

In this chapter, the utility function (UF) method will introduced and described. For a more detailed description, see either [66] or [68].

6.1 Brief description of the method

The utility function (UF) method is a biologically inspired arbitration method (see Chapter 5) for behavior selection in autonomous robots. One of the main purposes with the UF method is to provide a generally applicable method for behavioral organization in autonomous robots while, at the same time, minimizing the amount of hand-tuning of the parameters associated with the behavior selection mechanism, a common problem in most current methods for behavior selection [5]. Hand-tuning is avoided in the UF method by means of an EA that carries out evolutionary optimization of the selection mechanism in computer simulations (see Section 7.4). In the UF method, the selection of behaviors (described in Section 6.1.5) is based on the concept of **utility**, which is the topic of Section 6.1.1. Each behavior is associated with a **utility function** from which a scalar utility value can be obtained based on the current state of the robot, which, in turn, is specified by sensor readings and the values of internal variables (described in Section 6.1.3). Utility functions are further described in Section 6.1.4.

6.1.1 The concept of utility

In the UF method, utility is a scalar variable used for quantitative comparison of different behaviors, i.e. it provides the common currency in the behavior selection mechanism. The concept of utility has been studied in connection with ratio-

nal decision-making in both economics [4, 9, 65] and in ethology[1] [41, 40]. In ethology, utility is commonly referred to as **benefit**, or negative cost, and **rational behavior** in animals can be modeled as the result of the minimization of cost [40]. In economics, and in the UF method, rational behavior is associated with the maximization of utility. In fact, it has been shown by von Neumann and Morgenstern [65] that, under certain assumptions regarding the **preferences** of an individual (e.g. an animal or a robot), there exists a mapping from the set of possible outcomes $\{c_1, c_2, \ldots, c_n\}$ (resulting e.g. from the action taken in a given situation) to a scalar utility value $u(c_i),\ i = 1, \ldots, n$, such that $u(c_j) > u(c_k)$ if and only if the outcome c_j is preferred to the outcome c_k. A robot (or an animal) that consistently attempts to maximize utility, by selecting the appropriate behavior in any given situation, is said to exhibit rational behavior. It should be noted that rational behavior is not necessarily associated with **intelligent behavior**; even though a rational robot maximizes utility, its behavior will only be intelligent if its utility functions have been chosen in a suitable way. Thus, intelligent behavior requires properly shaped utility functions, which, in the UF method, are obtained through evolutionary optimization. The utility functions depend on a set of variables (see Section 6.1.3) that summarize the internal state of the robot as well as its current perception of the external world.

It should also be noted that rational behavior does not require reasoning [41, 63]. As an example, consider the chemotaxis of *E. Coli* bacteria that, despite their small size (making the detection of a spatial gradient impossible), are able to locate and stay in areas of high concentration of an attractant (e.g. food) using instead the *temporal* gradient of attractant concentration. It turns out (see e.g. [61] and [63]) that their chemotaxis can be accurately modeled as a simple maximization of utility, based on two very simple behaviors: *Straight-line movement* and *random tumbling*. Thus, despite an utter lack of reasoning capability, the *E. Coli* are able to find food very efficiently[2].

6.1.2 Behaviors

In the UF method, behaviors are divided into **task behaviors** and **auxiliary behaviors**. Even though all behaviors are assigned utility functions, only task behaviors, i.e. the behaviors that are directly related to the intended tasks of the robot, affect the fitness of the robot. The auxiliary behaviors generate no fitness

[1] The study of animal behavior in their natural environment.

[2] However, as noted in [61], it should also be added that this normally very useful navigation method turns out instead to be highly detrimental to the *E. Coli* if another type of bacterium (*M. Xanthus*) is present: Despite being much slower, the *M. Xanthus* are able to catch the *E. Coli* by releasing an attractant and then waiting, motionless, for their prey.

increase, but are nevertheless needed for the operation of the robot. For example, consider the case of a guard robot, whose task it is to patrol an industrial plant. In addition to task behaviors associated with the guarding of the plant, such a robot would need to be equipped with the auxiliary behavior of charging its batteries. While the robot is charging its batteries it cannot perform its assigned task, and thus the fitness increase is zero. However, if the robot fails to activate the charging behavior from time to time, it will forfeit all further fitness increases by coming to a halt for lack of energy. Thus, the *utility* of the charging behavior is non-zero, and increases as the energy level in the batteries decreases (assuming that the utility functions have been optimized and are able to provide appropriate utility values). Put differently, the robot's motivation for charging the batteries is increased. In simple applications, a single task behavior is usually sufficient.

6.1.3 State variables

The state of a robot is defined by all its sensing capabilities and the values of its internal variables. These variables are accessible to the behaviors and provide the only source of information. In the behavioral repertoire, each behavior is associated with a utility function that has a specific set of dependent variables. In the UF method, these variables are called **state variables** (denoted $\mathbf{z}$). There are three different kinds of state variables defined in the UF method, namely (1) **external variables** ($\mathbf{s}$), related to external sensors such as proximity sensors or bumper switches, (2) **internal physical variables** ($\mathbf{p}$), related to a robot's internal sensors such as the measurement of the battery level, and (3) **internal abstract variables** ($\mathbf{x}$), which roughly correspond to signaling substances (e.g. hormones) in biological systems. The general form of the N utility functions is thus given by

$$U_i = U_i(\mathbf{z}_i) = U_i(\mathbf{s}_i, \mathbf{p}_i, \mathbf{x}_i),\ i = 1, 2, \ldots, N, \tag{6.1}$$

where the subscript i denotes a set of variables associated with the i^{th} behavior.

The set of variables is defined by the user and may contain any variable available in the system. A robot can, of course, be equipped with any number of sensors (internal or external), and the signal obtained from a given sensor can be used for defining one or several state variables, depending on the complexity of the sensor. The user may also add internal abstract variables, the variation of which must then also be specified (or evolved), see e.g. [66] and Paper X. The exact composition of the set of state variables used in the utility functions is problem-dependent. For example, consider a case in which a robot is equipped with two external sensors (s_1, s_2), one internal physical variable (p_1), and one internal abstract variable (x_1). Furthermore, if the behavioral repertoire consists of two behaviors B_1 and B_2, the two associated utility functions (U_1 and U_2) may depend on any combination of

the *total* set of state variables $\{s_1, s_2, p_1, x_1\}$. The utility functions associated with B_1 and B_2 may, for example, be specified as

$$U_1 = U_1(s_1, s_2, p_1) \tag{6.2}$$

and

$$U_2 = U_2(s_2, x_1). \tag{6.3}$$

6.1.4 Utility functions

The general functional form of the utility functions, defined in Eq. (6.1), may be of any type. However, throughout this thesis and in [66], complete polynomials[3], i.e. polynomials that include all terms up to a certain degree, have been the preferred choice in the investigations. Since complete polynomials are used, the only parameter needed for the user to specify is the polynomial degree. Given the degree of a complete polynomial, the number of terms in each utility function is given by

$$N_t = \binom{n+d}{d} = \binom{n+d}{n}, \tag{6.4}$$

where n is the number state variables and d is the specified polynomial degree. This equation can be derived by introducing the trick of writing the product of variables in a given polynomial term $z_1^{p_1} z_2^{p_2} \cdots z_n^{p_n}$ as $z_1^{p_1} z_2^{p_2} \cdots z_n^{p_n} \times 1^{p_{n+1}}$, using the notation z_i for an arbitrary state variable (see Eq. (6.1))[4]. Letting

$$p_{n+1} = d - \sum_{i=1}^{n} p_i, \tag{6.5}$$

the problem is reduced to the combinatorial problem of placing (without replacement) d indistinguishable marbles in $m = n+1$ containers, neglecting the order in which they are placed in those containers. The number of ways (N_c) the marbles can be placed in the containers equals

$$N_c = \binom{m+d-1}{d}. \tag{6.6}$$

Replacing m by $n+1$ in the equation above results in Eq. (6.4). Consider, as an example, a utility function in the form of a complete polynomial with three state variables ($n = 3$) and $d = 2$. The function will thus have the following form

$$\begin{aligned} U(s,p,x) \;=\; & a_{000} + a_{100}s + a_{010}p + a_{001}x + a_{200}s^2 + a_{110}sp \\ & + a_{101}sx + a_{020}p^2 + a_{011}px + a_{002}x^2, \end{aligned} \tag{6.7}$$

[3] See also Paper VII for the definition of a complete polynomial.

[4] The constant term in the polynomial is represented by the combination $p_1 = \ldots = p_n = 0$.

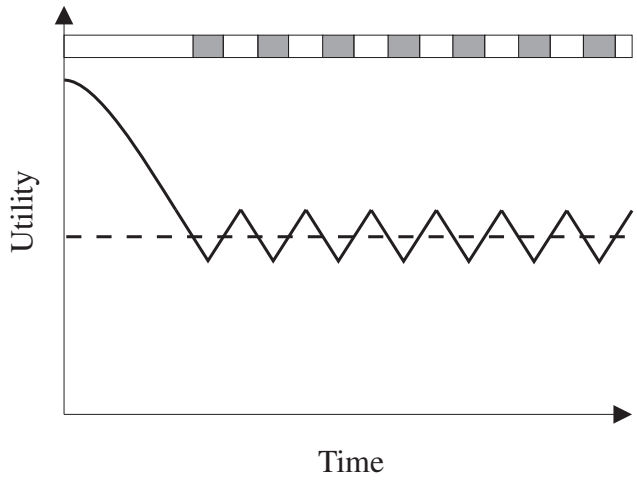


Figure 6.1: Illustration of behavior dithering, in which a robot oscillates between the behaviors B_1 and B_2, associated with the utility functions U_1 (solid line) and U_2 (dashed line). The horizontal bar at the top of the figure indicates the period of activation for the two behaviors (B_1 = white, B_2 = gray).

where a_{ijk} are constants to be determined by the EA. The number of terms equals 10, as expected from Eq. (6.4). Note that all terms are unique, as indicated by the subscript combination ijk in a_{ijk}, and that $i + j + k \leq d$.

All utility functions, and thus the behavior selection system, are optimized in simulations by means of an EA (see Section 7.4), where all coefficients (exemplified by a_{ijk} in Eq. (6.7)) are initialized to random values in a certain range.

6.1.5 Selection of behaviors

Once the utility functions have been defined, selecting a behavior for activation is simple. At all times, the behavior associated with the highest utility value is selected for activation. Hence, the behavior $B_{i_{\text{active}}}$ is selected according to

$$i_{\text{active}} = \arg\,\max(U_i)\ . \tag{6.8}$$

A common phenomenon encountered in arbitration methods is **behavior dithering** (behavior blending), i.e. a form of indecision. This phenomenon, illustrated in Fig. 6.1, occurs when a behavior is active during a very short time (up to a few time steps). Behavior dithering may cause unexpected overall behavior of a robot and is normally an unwanted feature. As a simple example, consider a robot equipped with the three behaviors *move forward* (B_1), *move backward* (B_2), and *stand still* (B_3). Since B_3 need not make use of the robot's motors (at least if the

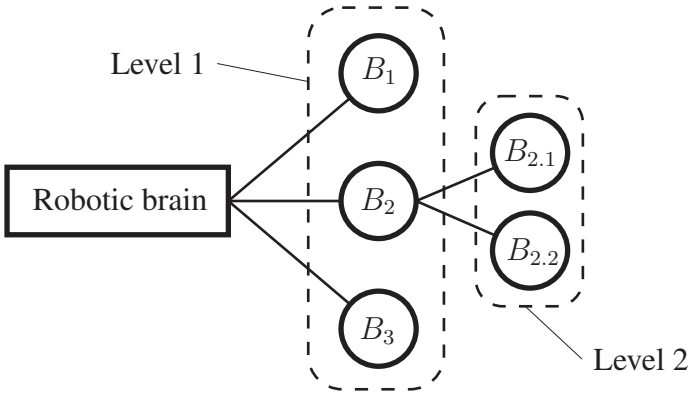


Figure 6.2: An illustration of a behavioral hierarchy consisting of five behaviors.

robot is statically stable) it is, of course, the preferred behavior for e.g. battery charging. However, a condition of standstill can also be achieved through rapid alterations between B_1 and B_2. However, in this case, the robot will certainly need to use its motors in a very inefficient way. In the UF method, behavior dithering can be avoided through the use of internal abstract variables (see Paper III for an example). Another way to reduce the risk of behavior dithering is to set the fitness function of the EA in such a way that fitness increases are associated only with the continuous execution of the task behavior, as was done e.g. in Paper VIII.

6.1.6 Behavioral hierarchies

Whenever possible, behaviors should be kept as simple as possible. Thus, if a behavior tends to become very complex, it should be decomposed into two or more behaviors, moving the complexity of the problem to the behavior *selection* system instead. Thus, one aspect of this process of subdivision is that it simplifies the implementation of behaviors. This is particularly important if the behaviors are implemented by hand, using e.g. the *if-then-else*-rules described in Section 2.1.1. A typical example is the *energy maintenance* behavior considered in Paper V and Paper VI. In those papers, this behavior was divided into two sub-behaviors; one responsible for finding the charging station and one that carried out the actual charging.

The structure resulting from behavioral decomposition defines the **behavioral hierarchy**, an example of which is illustrated in Fig. 6.2. In the UF method, the utility values of behaviors are compared on a level-by-level (breadth-first) basis, see Paper V. In the example illustrated in the figure, the hierarchy consists of five behaviors, on two different levels. Behaviors B_1–B_3 are compared first. If,

based on the utility values U_1–U_3, it happens that B_2 is selected for activation, the second level is traversed, where $B_{2.1}$ and $B_{2.2}$ are compared. In general, the hierarchy is traversed until a terminal (leaf) node is encountered.

It should also be mentioned that a further positive aspect of behavior decomposition is that it allows the use of behaviors in several different parts of the hierarchy.

Chapter 7

The utility function library

The UF method is implemented in a software library (hereafter referred to as UFLIB), currently consisting of around 50 source units with a total of around 12,000 lines of code. UFLIB is constantly being improved, as is the method itself. The main features of UFLIB are described in Paper V, and will only be mentioned briefly here. However, recently added functionality, such as the use of multiple simulations (see Paper VIII) and the implementation of the pathfinding algorithm, with an associated grid-based map (see Paper X), will be described in some detail. In addition, the implementation of two commonly used sensor types, namely infrared sensors and laser range finders, will be described.

Developed in object-oriented Pascal, UFLIB compiles both in Delphi [6] and with the FreePascal compiler [17]. However, 3D graphical visualization, generated using the GLScene graphics library [21], is currently not supported in FreePascal, and must thus be removed if that compiler is used. However, in this thesis, all simulations concerning the UF method have been carried out using UFLIB, compiled in the Delphi environment.

UFLIB is distributed in the form of a library rather than a stand-alone executable since, in the author's experience, it is very difficult, or even impossible, to try to generate a single simulation program, applicable to all types of problems. Instead, the approach taken when developing UFLIB has been to provide a basic library implementing the behavior selection mechanism, the evolutionary algorithm, the 3D visualization etc., while still allowing the user full freedom concerning e.g. the definition of the behavioral repertoire[1].

While the implementation of new behaviors requires that the user should write Pascal code (see also Section 3 in Paper V), most other features of UFLIB can be

[1] Some standard behaviors for wheeled robots, such as *straight-line navigation*, *obstacle avoidance*, and *locate IR beacon*, are provided with the library.

controlled in an easier way: In order to simplify the use of the library, particularly for users with limited programming experience, it has been written such that most features can be manipulated through simple text files, hereafter called **definition files**. In this chapter, a few examples will be given in the form of such files. Consequently, the syntax used in these files deserves an introduction. Since UFLIB is implemented using an object-oriented approach, the definition files follow a similar structure, in which each object is defined by its class name and its properties. The structure is illustrated in Listing 7.1, where each object starts with the keyword `object` and ends with the keyword `end`, and where properties of different types (e.g. floating point, boolean, string, vector, and matrix) are assigned their values. The names of these properties are defined by the user. In Listing 7.1, generic names have been used for the properties. Examples of object definitions are shown in Listing 7.2 and Listing 7.3.

Listing 7.1: General illustration of an object definition.

```
object <Name>: <ClassName>
  FloatProperty = 0.1
  BooleanProperty = True
  StringProperty = 'A string'
  VectorProperty = 1.0 2.0 3.0
  MatrixProperty =
    1.0 0.0
    0.0 2.0
end
```

7.1 Sensor modeling

In the current implementation of UFLIB, several sensor types are implemented, namely infrared sensors, laser range finders, battery sensors, wheel encoders, and beacon detectors. The battery sensor enables the monitoring of the energy level in a battery. Wheel encoders are used for counting pulses that are generated when a wheel axis rotates (in the case of a wheeled robot), thus allowing the robot to estimate its position through the use of odometry. Beacon detectors make it possible for the robot to detect a specific target (an infrared beacon), and have mainly been used for the localization of charging stations (see Paper V, Paper VI, and Paper VIII). All sensor types implemented in UFLIB support Gaussian noise modeling. In the following subsections the detailed implementation of two important sensor types, namely the **infrared sensor** and the **laser range finder**, will be described.

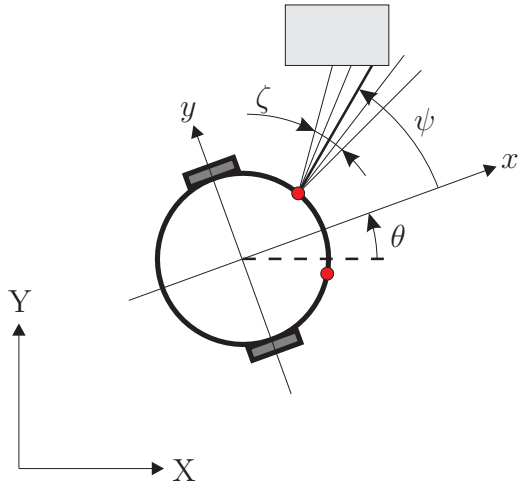


Figure 7.1: A differentially steered robot equipped with two infrared sensors (red filled circles). ψ is the direction of the sensor, relative to the robot's heading θ. The sensor's opening angle is denoted ζ. Lines emanating from the sensor indicate the rays used in the calculation of the sensor reading. Here, five rays are shown of which three intersect the obstacle (light gray rectangle). The global (inertial) coordinate system is denoted by capital letters (XY) whereas small letters (xy) denote the robot's coordinate system.

7.1.1 Infrared sensor

An infrared (IR) sensor may be modeled in a variety of ways. The implementation in UFLIB uses a ray tracing technique, illustrated in Fig. 7.1, in which a sensor's reading is obtained as a sum of contributions from a number of rays. Each ray is used for calculating the distance to the nearest object by searching for intersections between the ray and objects in the environment (arena). This process takes place in a two-dimensional horizontal slice of the arena, taken at the height at which the sensor is positioned. Since a slice of the arena generates a number of lines, the process of finding intersections amounts to a search for intersections between line pairs in the horizontal plane, where one line represents a ray emanating from the sensor and the other represents (a part of) an object in the arena. An example of a sliced arena can be seen in the right panel in Fig. 7.2. If an intersection between a ray and an object in the arena is found, the distance to the intersection point is calculated. This procedure is repeated for every ray in the sensor model. The complete sensor reading is then formed as a sum of contributions from the N_r rays as

$$s = \frac{1}{N_r} \sum_{i=1}^{N_r} \rho_i, \tag{7.1}$$

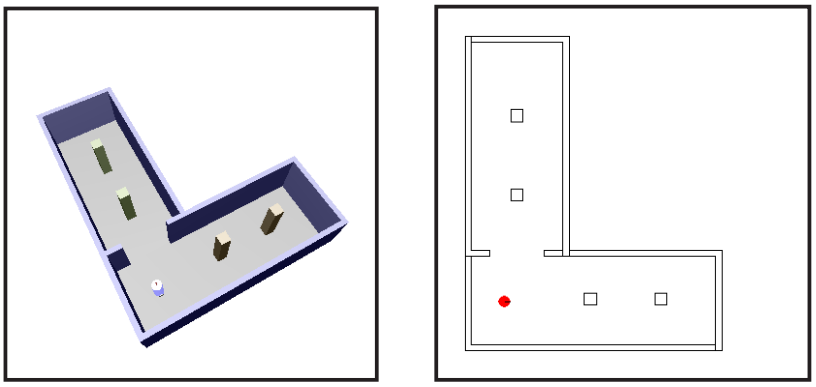


Figure 7.2: Example of a sliced arena, similar to the one used in Paper VI. The original arena is shown in the left panel and the right panel shows the arena sliced at a certain height. A robot is also shown in the figure. It appears as a red filled disk in the right panel.

where ρ_i is the contribution from the i^{th} ray, calculated according to the empirical equation (see also [29])

$$\rho_i = \begin{cases} \min\left(\left(\frac{\alpha}{d_i^2} + \beta\right)\cos\kappa_i, 1\right), & \text{if } d_i \leq R \\ 0, & \text{otherwise} \end{cases} . \tag{7.2}$$

where d_i is the distance to the nearest object along the i^{th} ray, R is the range[2] of the sensor, α and β are constants whose values are set in an empirical manner, and

$$\kappa_i = -\frac{\zeta}{2} + (i-1)\delta \ \in [-\pi/2, \pi/2] \tag{7.3}$$

is the angle of ray i, relative to the direction of the sensor (ψ). The rays are enumerated in ascending order, starting from 1 and increasing in the direction of ψ, i.e. the angle of the first ray (relative to the robot's coordinate system) is equal to $\psi - \zeta/2$. The parameter ζ represents the opening angle of the sensor, illustrated by the sector covered by the 5 rays emanating from the sensor in Fig. 7.1, and δ is the angle between adjacent rays, calculated as

$$\delta = \frac{\zeta}{N_r - 1}. \tag{7.4}$$

The angle of the i^{th} ray, with respect to the inertial coordinate system (shown as XY in Fig. 7.1), is defined as

$$\gamma_i = \theta + \psi + \kappa_i, \tag{7.5}$$

[2]A typical range of an IR sensor is up to a few decimeters.

where θ is the robot's heading and ψ is the relative angle of the sensor. It should be noted that the reading of a real IR sensor is generally quite diffuse. Thus, the reading s should not be considered as an exact measure of (inverted) distance, but rather as a qualitative measure of the presence or absence of an object in front of the sensor. Noise can be added to the sensor reading as

$$s \leftarrow s\, N(1, \sigma), \tag{7.6}$$

where $N(1, \sigma)$ is a normal-distributed value with mean 1 and standard deviation σ. As a final step, in order to ensure that the sensor model generates a properly bounded value, the sensor reading is adjusted according to

$$s \leftarrow \max(\min(s, 1), 0)\ . \tag{7.7}$$

All parameters mentioned above can easily be accessed via the robot's definition file. As an example, the definition of an IR sensor is shown in Listing 7.2.

Listing 7.2: Definition of an infrared sensor.

```
object IRSensor: TIRSensor
  RelativePosition = 0.1000 0.1732 0.3000
  RelativeDirection = 0.0000 0.0000 1.0472
  OpeningAngle = 0.5000
  Range = 0.3000
  NumberOfRays = 5
  Alpha = 0.0300
  Beta = 0.1000
  NoiseLevel = 0.0100
end
```

7.1.2 Laser range finder

A laser range finder (LRF) is a device that measures distances to objects in the arena by means of laser light and time-of-flight calculations. LRFs are much more expensive than IR sensors, but commonly provide very accurate distance measurements, as opposed to the diffuse readings obtained from IR sensors. LRFs are also less sensitive to changes in e.g. lighting conditions than IR sensors. Using a rotating lens and mirror system, an LRF is able to scan a wide sector in a short period of time. LRFs are capable of accurate measurement of distances up to several meters, or (for some types) even kilometers [54].

The LRF model implemented in UFLIB has been based on a 2D laser range finder (URG-X002S) manufactured by Hokuyo [26]. This LRF shown in Fig. 7.3, has a range of 4.0 m, an angular resolution of $360°/1024$, and a $270°$ wide scanning sector. Before the LRF was implemented in UFLIB, the author conducted

Figure 7.3: A laser range finder (URG-X002S) manufactured by Hokuyo. Photo by the author.

Table 7.1: Data sampled from the laser range finder shown in Fig. 7.3. The setup of the experiment is illustrated in Fig. 7.4. Summarized in this table are the actual distances (x), the average of 10 distance measurements $(\bar{x})$, the standard deviation (σ), and the systematic distance error $(\bar{x} - x)$.

x [m]	$\bar{x}$ [m]	σ [mm]	$\bar{x} - x$ [m]
0.25	0.29	1.26	0.04
0.50	0.53	0.92	0.03
0.75	0.79	1.27	0.04
1.00	1.06	1.08	0.06

experiments on an actual Hokuyo LRF. Some results from those experiments are summarized in Table 7.1 and the experimental setup is shown in Fig. 7.4. The implementation of the LRF in UFLIB is similar to that of the IR sensor, described in Section 7.1.1. The main difference between the IR sensor and the LRF is that, in the model of the LRF, rays are not weighed together to form a single value. As shown in the middle panel of Fig. 7.5, in the LRF, a number of uniformly distributed rays are generated in the horizontal plane at the height of the LRF. Normalized distances to intersections between the rays and the lines representing the objects in the arena are taken as the readings of the LRF. Thus, an LRF will have as many readings as the number of rays specified in its definition file, shown in Listing 7.3. Each reading will have a value in the range $[0, 1]$ or, in case a particular ray does not produce a reading, a value of -1. In addition, and as seen in Listing 7.3, it is possible to specify the sector scanned by the LRF by means of a

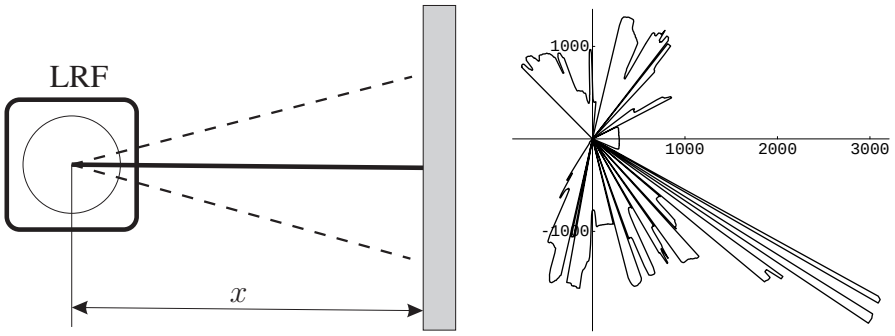


Figure 7.4: Experimental setup (left panel) used in the measurements of the laser range finder (LRF) shown in Fig. 7.3. In this experiment, the ray emanating from the center of the LRF and reaching the obstacle (gray rectangle) at a right angle was sampled and the associated reading was recorded for four different values of x. For each value of x, 10 measurements were made in order to form an average and to calculate an estimated standard deviation. The results from these measurements are summarized in Table 7.1. Dashed lines indicate rays not considered in this particular experiment. The right panel shows the actual reading of the LRF in one of the measurements ($x = 0.25$ m), i.e the distance to the nearest object in all covered directions, obtained by the LRF. The numbers on the axes in the right panel are given in millimeters.

maximum and minimum angle relative to the robot's heading.

In the LRF implemented in UFLIB, noise is added to the LRF readings, using the same procedure as for the IR sensors, i.e. the distance obtained for each ray is slightly modified as

$$r_i \leftarrow r_i\, N(1, \sigma),\ i = 1, 2, \ldots, N_r, \tag{7.8}$$

where N_r is the number of rays and r_i is the i^{th} ray reading. Readings including noise are shown in the right panel in Fig. 7.5, in which dots indicate the points of intersection.

Listing 7.3: Definition of a laser range finder.

```
object LaserRangeFinder: TLaserRangeFinder
  NumberOfRayAngles = 128
  RelativePosition = 0.1500 0.0000 1.0000
  RelativeDirection = 0.0000 0.0000 0.0000
  MinimumRelativeAngle = -1.0472
  MaximumRelativeAngle =  1.0472
  SweepDirection = 1
  Range = 4.0000
  NoiseLevel = 0.0010
end
```

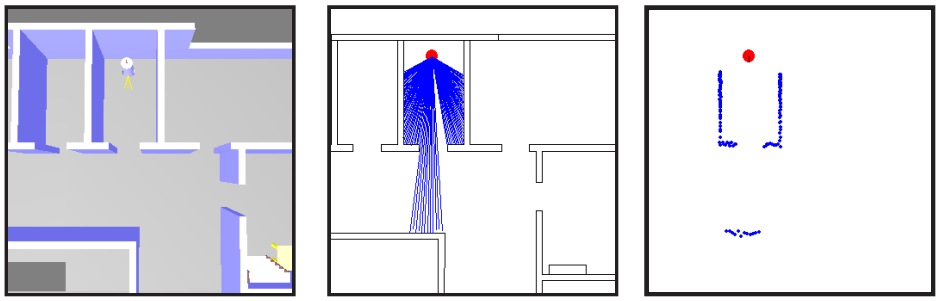


Figure 7.5: An illustration of simulated laser range finder (LRF) readings obtained in a typical arena. For clarity, only 128 rays are shown and the sector is limited to the range $[-60°, 60°]$. The left panel shows the arena (with the robot) from above. In the middle panel, the 128 rays emanating from the LRF are shown as lines. The right panel depicts the points sampled by the simulated LRF. The absence of lines in certain directions in the middle panel is due to the limited range of the LRF.

7.2 Motor modeling

One of the most common motors used in robot applications is the direct current (DC) motor, and this is also the actuator type implemented in the current version of UFLIB. The equivalent circuit, used for modeling the DC motor, is shown in Fig. 7.6, where V is the applied terminal voltage, R is the electrical resistance of the motor windings, L is the electrical inductance, and V_{EMF} is the back EMF counteracting V. V_{EMF} is proportional to the rotational velocity of the motor, i.e.

$$V_{\text{EMF}} = K_e \omega, \tag{7.9}$$

where K_e is the electrical constant and ω is the angular velocity of the motor's shaft. The electrical model takes the form

$$V = L\frac{\mathrm{d}i}{\mathrm{d}t} + Ri + V_{\text{EMF}}, \tag{7.10}$$

where i is the armature current. In robot applications, the DC motor is commonly used for actuating mechanical parts. Such parts normally have a much larger time constant than the electrical part, and the inductance term can therefore be neglected. By neglecting the inductance term and using the fact that the armature current is proportional to the torque generated by the motor ($\tau_g = K_t\, i$), Eq. (7.10) in combination with Eq. (7.9) yields

$$V = R\frac{\tau_g}{K_t} + K_e\, \omega, \tag{7.11}$$

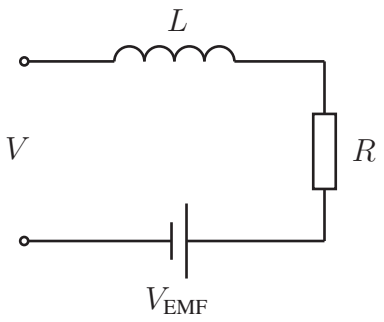


Figure 7.6: Electrical circuit representing a DC motor.

where K_t is the motor's torque constant. By rearranging Eq. (7.11), the generated torque can be expressed as

$$\tau_g = \frac{K_t}{R}(V - K_e\,\omega). \tag{7.12}$$

As is common in devices that involve moving mechanical parts, losses due to friction must be accounted for. Thus, the final torque is modified according to

$$\tau = \tau_g - K_c\,\mathrm{sgn}(\omega) - K_v\,\omega, \tag{7.13}$$

where $K_c\,\mathrm{sgn}(\omega)$ is the loss due to Coulomb friction and $K_v\,\omega$ is the loss due to viscous friction. The constants K_c and K_v are normally set in an empirical manner.

Due to their low torques and high angular velocities, DC motors are usually accompanied by a gear box. In UFLIB, the gear box is implemented as a loss-free transformation of the motor output. Thus, assuming loss-free transmission ($\tau\omega = \text{constant}$), the output from the gear box is calculated according to

$$\tau_{\text{out}} = G\tau, \tag{7.14}$$

$$\omega_{\text{out}} = \frac{1}{G}\omega, \tag{7.15}$$

where G is the gear box ratio. Losses in the gear box are added as

$$\tau_{\text{out}} \leftarrow \tau_{\text{out}} G_e, \tag{7.16}$$

where $G_e \in [0, 1]$ is the gear box efficiency. As a final step, noise is added to the output torque according to

$$\tau_{\text{out}} \leftarrow \tau_{\text{out}} N(1, \sigma), \tag{7.17}$$

where $N(1, \sigma)$ is a normal-distributed value with mean 1 and standard deviation σ. All parameters associated with a DC motor can be accessed through the motor's definition file, an example of which is given in Listing 7.4.

Listing 7.4: Definition of a DC motor.

```
object Motor: TDCMotor
  MaximumVoltage = 12.0000
  TorqueConstant = 0.0333
  BackEMFConstant = 0.0333
  ArmatureResistance = 0.6200
  CoulombFriction = 0.0080
  ViscousFriction = 0.0200
  GearRatio = 40.0000
  GearEfficiency = 1.0000
  MaxTorque = 0.3000
end
```

7.3 Recent additions

As mentioned in the beginning of the chapter, UFLIB is still under development, with novel features being added continuously. In this section, two such features will be described in some detail, namely the use of multiple simulations and a pathfinding procedure.

7.3.1 Multiple simulations

An important issue in evolutionary robotics (ER) is adaptation to special conditions (overfitting), a problem that plagues any ER investigation in which the performance evaluation of robots is based on results obtained in simulations starting from a given, single initial condition. In order to mitigate such problems, the concept of **multiple simulations** has recently been added to UFLIB.

In this case, and as described in Paper VIII, the **evaluation** of a robot consists of a set of simulations, where each simulation belongs either to a **training set** or a **validation set**. During a run of the EA, only the simulations in the training set determine the overall fitness of an individual (i.e. a robot). The validation simulations are conducted only under certain circumstances, for example when an individual with higher **overall fitness** (as obtained from the training simulations) than the previous best individual is found.

When multiple simulations are used, a robot is subjected to many different situations during training, making it less likely that it would gain high overall fitness by adapting to special conditions typical of *one* specific simulation. Each simulation in the training set produces a separate fitness value. In order to generate an overall fitness value, the performance measures must be combined into a single,

scalar value. In UFLIB, the following three combined fitness measures are used:

$$F_{\text{avg}} = \frac{1}{N_T}\sum_{i=1}^{N_T} f_i, \tag{7.18}$$

$$F_{\text{min}} = \min\left(f_1, f_2, \ldots, f_{N_T}\right), \tag{7.19}$$

$$F_{\text{eps}} = F_{\text{min}} + \varepsilon F_{\text{avg}}, \tag{7.20}$$

where f_i is the fitness achieved in the i^{th} training simulation, ε is a small positive constant, and N_T is the number of training simulations. For a comparison of the different measures, see Paper VIII and Chapter 9.

Changing the fitness measure amounts to editing the value assigned to the property `FitnessMeasureType` (see Listing 7.5) in the file that defines the evaluation. In the current implementation of UFLIB, the type of fitness measure may take the values `'fmtAverage'`, `'fmtMinimum'`, or `'fmtMinEpsAvg'` (see Eqs. (7.18), (7.19), and (7.20), respectively). In addition, the user may also define a custom fitness measure (`'fmtCustom'`).

The property `UseInFitnessMeasure` decides whether a simulation is included in the training set (`'True'`) or in the validation set (`'False'`). An example of the definition of an evaluation is shown in Listing 7.5. For brevity, this example only defines a single simulation for each set. However, there is no upper bound on the number of simulations in each set. The minimum requirement is that the training set should contain at least *one* simulation.

7.3.2 Pathfinding

In many navigation tasks in which a map is available, finding a path between two locations is a useful feature, if it can be achieved fast and in a reliable manner. A recently added part of UFLIB is the automatic generation of a 2D **grid map**, based on the 3D environment. The size of the grid cells is based on the size of the robot, as illustrated in Fig. 7.7. When the grid map is generated, it is ensured that at least four grid cells fit into the 2D projection of the robot's axis-aligned bounding box[3]. In this way, sufficient resolution is ensured while maintaining the speed of the **pathfinding** algorithm. The algorithm currently implemented in

[3] An axis-aligned bounding box, indicated by the rectangle around the robot in Fig. 7.7, is the smallest volume that encapsulates the robot and whose sides are aligned with the axes of the global coordinate system.

Listing 7.5: Definition of an evaluation consisting of two simulations; one used for training and the other for validation. See the main text for a detailed description.

```
object Evaluation: TEvaluation

  FitnessMeasureType = 'fmtMinimum'

  object TrainingSimulation: TUFRobotSimulation
    InitialPosition =  0.20  0.20  0.00
    InitialDirection = 0.00  0.00  0.60
    UseInFitnessMeasure = True
    SimulationTime = 100.00
    TimeStep = 0.01
  end

  object ValidationSimulation: TUFRobotSimulation
    InitialPosition =  0.10  0.10  0.00
    InitialDirection = 0.00  0.00 -0.10
    UseInFitnessMeasure = False
    SimulationTime = 100.00
    TimeStep = 0.01
 end

end
```

UFLIB uses an A* search algorithm for finding the shortest path (or, in general, the path associated with the lowest cost (see Eq. (7.21) below) between a starting point and a target location (see the example code in Listing 7.6).

The A* algorithm [24] is one of the most common general search algorithms, and also one of the fastest. A* uses two lists, an *open* list and a *closed* list. The open list is used for maintaining a set of nodes that have not yet been examined (updated) and the closed list contains a set of nodes that have already been examined. Initially, the closed list is empty and the open list contains the node at which the search starts. A node n maintains information about the cost of going from the initial node $\mathcal{A}$ to n and an estimate of the cost for moving from n to the goal node $\mathcal{B}$. Using standard notation, the combined cost of a node is calculated as

$$f(n) = g(n) + h(n), \tag{7.21}$$

where $g(n)$ is the cost of moving from $\mathcal{A}$ to n and $h(n)$ is the estimated cost of moving from n to $\mathcal{B}$. $h(n)$ is commonly referred to as the heuristic of the algorithm. Common heuristics involve the Euclidean distance, which is used in UFLIB, and the Manhattan distance. The Euclidean distance between two points

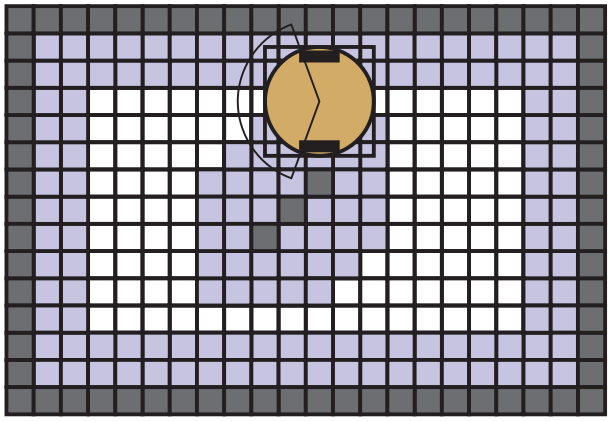


Figure 7.7: Illustration of a grid-based map used by the A* pathfinding algorithm. Colored cells indicate either occupied areas (dark gray) or cells treated as occupied in order to provide sufficient clearance for the robot to avoid collisions with the static objects during navigation. Only white cells are considered as potential waypoints in the A* algorithm. The size of the cells is automatically calculated based on the robot's axis-aligned bounding box (indicated by the rectangle surrounding the robot) in such a way that this box will contain at least 16 cells. Margins, marked in light blue color, are automatically added, in two steps, in order to generate sufficient clearance.

in the plane $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ is defined as $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, whereas the Manhattan distance is defined as $|x_1 - x_2| + |y_1 - y_2|$.

It can be shown that the shortest path will be found (assuming that a path exists), provided that $h(n)$ does not overestimate the distance [56]. If $h(n)$ is overestimated, the search might become faster but there is no guarantee that the algorithm will find the shortest path. Once the algorithm has finished, the final, and hopefully shortest, path is generated by a process of backtracking, in which the trail of the lowest cost ($f(n)$) is followed from $\mathcal{B}$ to $\mathcal{A}$. For a more detailed description of the A* algorithm, see [24, 56].

In Fig. 7.8, the generation of a grid map is illustrated by means of the arena used in Paper VIII. In the right panel of the figure, the path from A to B, as generated by the A* algorithm, is shown in dark gray color. The result from the pathfinding algorithm is a sequence of **waypoints**, each located in the center of a grid cell, as indicated in Fig. 7.9. A path-following robot travels from A to B by navigating through each waypoint in the generated sequence. Thus, provided that the problem of localization can be solved (see Paper X), the robot will be able to travel the entire distance based on the generated sequence of waypoints. In order to avoid generating a path that may cause the robot to collide with static obstacles in the environment, a clearance margin of two grid cells is added before the path

Listing 7.6: Example code for generating a gridmap.

```
procedure Test;
var
  Arena: TArena;
  Robot: TRobot;
  GridMap: TGridMap;
  XWaypoints, YWaypoints: TVector;
begin
  // Create instances of an arena, a robot, and a gridmap.
  Arena := TArena.CreateFromFile('MyArena.txt');
  Robot := TRobot.Create;
  Robot.LoadFromFile('MyRobot.txt');
  GridMap := TGridMap.Create;

  // Generate the gridmap.
  GridMap.Generate(Arena, Robot.Body);

  // Generate a path from (0,0) to (1,2).
  GridMap.FindPath(0.0, 0.0, 1.0, 2.0)

  // Retrieve the waypoints.
  XWaypoints := TVector.Create;
  YWaypoints := TVector.Create;
  GridMap.GetPathAsWayPoints(XWaypoints, YWaypoints);

  ...

end;
```

is generated. This provides a sufficient amount of clearance (again assuming accurate navigation) between the robot and the obstacles. The margin is indicated by light gray color in the right panel in Fig. 7.8 (see also Fig. 7.7). Only white cells are considered as potential waypoints by the pathfinding algorithm.

7.4 Evolutionary algorithm

The EA used in UFLIB for the optimization of utility functions operates on a population of individuals (robots), and follows the general procedure outlined in Algorithm 3.1 (see also Listing 7.7). An example of a chromosome encoding the N utility functions of an individual is shown in Fig. 7.10. As is indicated in the figure, the genes are quite complex and encode an entire utility function polynomial each. Each individual is evaluated by running the N_T simulations

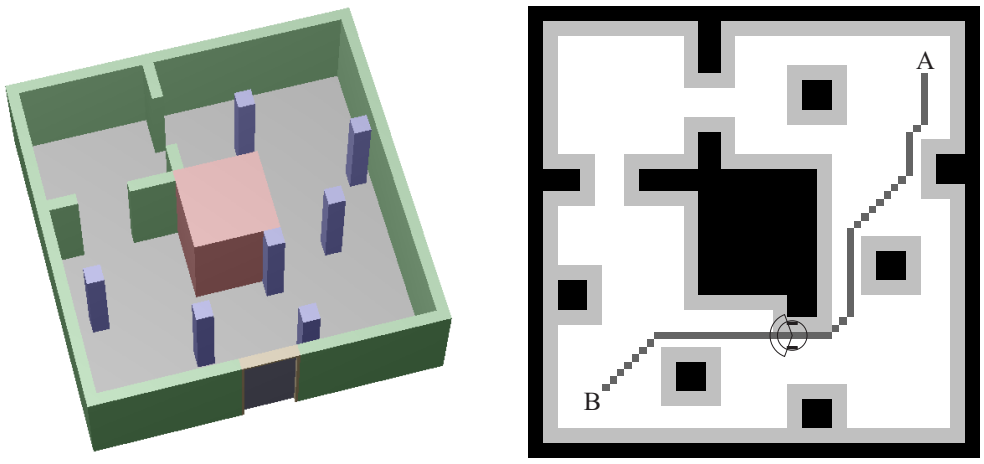


Figure 7.8: Discretization of an arena (left panel) into a grid map (right panel) that can be utilized by the A* pathfinding algorithm. Black cells indicate static obstacles and gray cells indicate the added margins. In this snapshot, the robot, in its true size, is shown halfway along the path going from A to B.

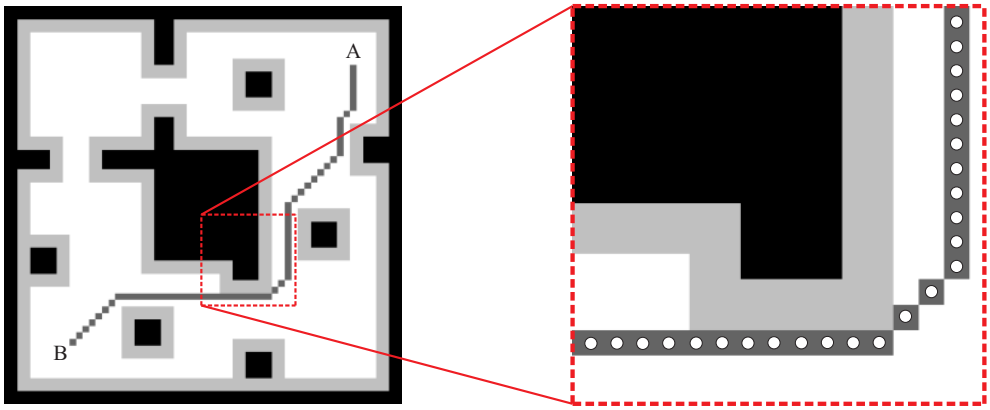


Figure 7.9: Illustration of the placement of waypoints in a grid-based map. In the right panel, a magnification is shown in which the waypoints are illustrated by white circles. Note that each waypoint is placed in the center of the grid cell.

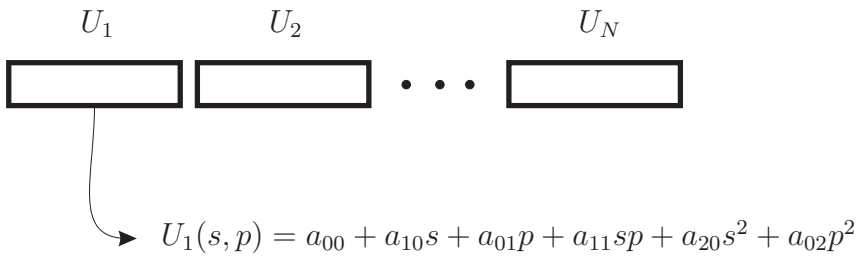


Figure 7.10: Example of a chromsome in which N utility functions are encoded. In UFLIB, each gene encodes an entire utility function. Thus, there is a total of N genes. Here, a decoded version of the first gene is illustrated as a second degree polynomial having two state variables (s, p).

constituting the training set, as defined by the corresponding definition file. The EA uses generational replacement, i.e. a new population is formed that replaces the previous one in each iteration (generation) of the EA. During the formation of the new population, the EA performs a sequence of selection, crossover, and mutation. The default crossover operator, illustrated in the top panel of Fig. 2 in Paper VII, simply swaps a randomly chosen sequence of genes between two selected chromosomes. Thus, it effectively swaps entire utility functions. The mutation operator modifies the coefficients, shown as a_{ij} in Fig. 7.10, in a random manner. The crossover and mutation operators are applied with a certain probability, set by the user. It should be noted that it is possible for the user to implement modified versions of the crossover and mutation operators. However, the standard operators are usually sufficient.

7.5 Usage example

In this section, the basic steps that a user of UFLIB must perform in order to build an application for evolutionary optimization of a behavior selection mechanism are given. A more detailed description of the various parts that should be specified is given in Paper V. Complementary material in the form of a reference manual and a tutorial, including template files, are available for download at [70]. In addition, a demonstration program is available for download at [69].

Although one of the main aims of this software library is to minimize the amount of work that a user must carry out to develop software for scientific investigations involving autonomous robots, many parts of a study remain specific to the problem at hand. In any investigation involving behavior selection in autonomous robots, the first step involves the implementation of the constituent behaviors included in the behavioral repertoire. Behaviors may be implemented using any

Listing 7.7: Main loop for the evaluation of a generation (cf. Algorithm 3.1). The method `FinalizeFitness` (belonging to the object `Evaluation`) generates a fitness measure for the evaluated individual according to one of the equations (7.18), (7.19), and (7.20).

```
repeat
  for i := 1 to EA.Population.Size do
  begin
    Robot := EA.Population[i];
    Robot.DecodeGenome;
    for j := 1 to Evaluation.NumberOfSimulations do
    begin
      RS := TRobotSimulation(Evaluation[j]);
      RS.SetAgent(Robot);
      RS.SetArena(Arena);
      RS.Run;
    end;
    Evaluation.FinalizeFitness;
    Robot.Fitness := Evaluation.Fitness;
  end;
  EA.MakeNewGeneration;
until (TerminationCriteriaFulfilled);
```

architecture, for instance those described in Section 2.1. However, since UFLIB provides a general framework for behavior selection, the choice of behavioral architectures is not limited to the ones treated in this thesis — any implementation of a behavior may be used, as long as it complies with the interface defined by UFLIB. The actual implementation requires that the user should define each behavior in the form of Pascal source code (unless only the behaviors provided with UFLIB are used). Using templates, the writing of source code can be reduced to the specification of a few procedures.

As a second step, and as required in all optimization procedures involving evolutionary methods, a suitable fitness measure must be defined. As mentioned in Section 6.1.2, this measure is normally associated with the execution of a single behavior (the task behavior). Although a single task behavior is sufficient in most situations, multiple task behaviors may be defined as well.

The third step involves the definition files for the simulated robot, the EA and its parameters, and the evaluation setup. In UFLIB, a robot consists of two parts, namely a *body* and a *brain*, both being specified in the robot's definition file. The body consists of specifications of the physical parameters of the robot (such as e.g. its height, weight, and shape, as well as its motors and sensors), whereas the brain consists of a specification of the behaviors included in the behavioral

repertoire and the set of state variables (see Section 6.1.3) associated with each behavior. In addition, the behavioral hierarchy must be defined by the user (see Section 6.1.6 for an example).

In the definition file associated with the EA, parameters such as population size, crossover probability, and mutation probability should be specified. The user also has the option to specify a custom crossover operator. The definition file for the evaluation setup (see Section 7.3.1) defines the simulations constituting the training set and the validation set. For each simulation, parameters related to the maximum allowed simulation time, initial position and heading, and the time step used by the numerical solver, must be defined. In addition, the choice of the composite fitness measure, defined by Eqs. (7.18)–(7.20), should be made.

The final step involves the arena definition file that defines the environment in which the robot is to be evaluated. This file consists of a list of objects such as walls and other obstacles. A more complete description of the available objects can be found in [70].

Once the steps just listed have been completed, the actual application, possibly with a graphical user interface, can be created rapidly (using e.g. a template associated with UFLIB) and compiled, and the evolutionary optimization of the behavior selection mechanism can commence (see also Paper V).

Chapter 8

Case studies

In this thesis, a few different cases of behavior generation have been studied, namely in Paper I, Paper III, and Paper IV. In Paper I and Paper IV, bipedal gaits were generated for a simulated two-legged robot. In Paper III, both the generation and organization of behaviors were investigated using simulations of a one-legged hopping robot. The problem of behavioral organization has been investigated in several different cases, involving a simple guard robot (Paper VI), an exploration robot (Papers V, VII, and VIII), and, more recently, a transportation robot (Papers IX and X). In this chapter, two cases will be described in some detail, namely those studied in Paper III and in Paper X. The first case has been selected for inclusion in this chapter since it illustrates a limitation of behavior selection, namely that, regardless of the quality of the behavior selection mechanism, the overall performance of the robot will be limited by the quality of the individual behaviors constituting the behavioral repertoire. The second case merits special consideration since it will be the first application of the UF method in a complex, real-world task.

8.1 Hopping robot

In Paper III, the problem of behavioral organization was investigated using the UF method (see Chapter 6). The behaviors, namely *move forward*, *move backward*, *stop*, and *charge batteries*, were represented as RNNs and were used for controlling a one-legged hopping robot (see Fig. 2 in Paper III). The robot was thus simpler than a bipedal robot, but still exhibited highly non-trivial dynamics. While a one-legged robot is unlikely to be used in real-world applications, it was sufficient for this study, the aim of which was to study the generation and selection of behaviors. The investigation was performed using a two-stage process: First the

constituent behaviors were evolved, then the behavioral organizer was evolved, in both cases by means of an EA. During the optimization of the behavioral organizer, the constituent behaviors remained fixed. The behavioral organizer (optimized by the EA) performed satisfactorily, activating the various behaviors at appropriate times. However, the RNNs representing the behaviors did not always perform in a satisfactory manner when activated in situations that were different from the ones used when they were generated. For instance, the *move forward* behavior was evolved with the robot starting from a standstill upright position. After the optimization, the robot was able to travel forward, in a slightly curved path, for the entire length of the simulation. However, when used during the optimization of the behavioral organizer, the behavior was less reliable. As the behavioral organizer may activate a behavior at any time, the probability that the initial state of the robot will differ from that used during training is large. In fact, it is highly unlikely that a moving one-legged robot will find itself in the *exact* same dynamical state (i.e. with the same heading and speed) more than once at the very instants when the behavior is activated.

Even though the behavior selection mechanism was somewhat constrained by the weak performance of the constituent behaviors, the UF method still managed to solve the problem of behavior selection in a satisfactory manner. The success of the method was verified in a simplified setup, in which the robot was represented by a simple solid body, sliding (rather than hopping) along the surface.

8.2 Transportation robot

Internal transportation of goods in e.g. hospitals and factories is an important but tedious task, which, in many cases, could very well be carried out using autonomous robots [14, 62, 64]. In Paper X, the aim is to develop a general-purpose robot to be used for internal transportation tasks. An outline of this project is given in Paper IX. The transportation robot is a wheeled robot, intended for operation in dynamic environments. In some of the currently available robotic transportation systems [2], the environment must be adapted to the robot (e.g. through the installation of laser navigation devices such as reflective targets). By contrast, for the transportation robot developed in the project described in Papers IX and X, no modification of the arena should be needed.

In order to move reliably from a pickup position to a delivery site, the robot must use a number of sensor modalities in order to achieve an accurate estimate of its relative position in the environment. In addition, the robot must be equipped with a communication interface through which a user can give commands. For localization purposes, the robot is to be equipped with a map of the environment,

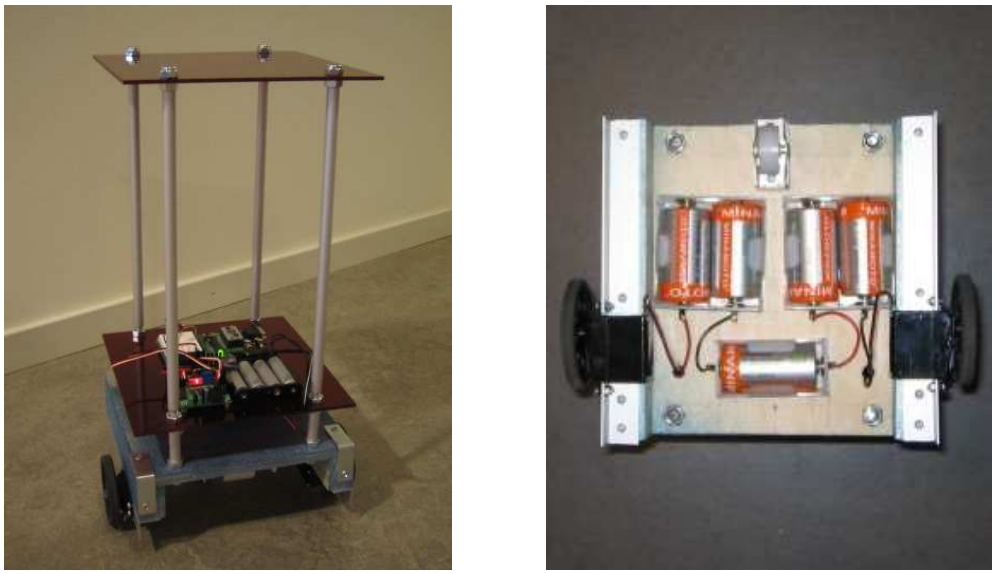


Figure 8.1: A prototype (scale 1:3) of the transportation robot. The right panel shows the battery pack mounted underneath the robot.

either given to it through the communication interface or generated through exploration of the environment. Also, in order to locate delivery sites and to calibrate the odometry, the robot should be equipped with a number of behaviors that perform actions such as laser scan matching based on the readings of a laser range finder mounted on the robot (see Papers IX and X), as well as landmark detection using a combination of all available sensor information in order to make the localization as accurate as possible. Due to the unpredictable nature of the environment in which the robot will operate, other behaviors such as obstacle avoidance and alternative path generation are also needed.

Autonomous robots capable of performing tasks such as the one indicated above will probably become more common in the future. Indeed, already today, some robots have been developed for this purpose (e.g. [20, 51]). However, the general problem of reliable behavior selection in such complex robots is interesting in its own right and should be investigated further. One of the main goals in this transportation robot project is thoroughly to test and evaluate the robustness of the UF method and also its ability to generalize. Since the UF method allows for an expansion of the behavioral repertoire, the operational space of the robot can be extended, and the UF method can thus be evaluated in scenarios of ever-increasing complexity. Currently (August 2006), the simulations (based on UFLIB) of a slightly simplified version of the transportation robot are being carried out.

The robot depicted in the left panel of Fig. 8.1 is a simple first prototype, developed during the summer of 2006, for the transportation robot. This prototype has been built for real-world verification of the simulations, and, in particular, to test some of the behaviors required by the transportation robot, namely *navigation*, *obstacle avoidance*, and (slightly simplified) *localization*. The prototype will also be used for evaluating different hardware configurations, including (but not limited to) the choice of microcontroller and the system for localization.

The prototype depicted in Fig. 8.1, with a cross section of size 0.20×0.20 m, and a height of 0.42 m, is a 1:3 scale model of the actual transportation robot for which construction will begin in the spring of 2007. The robot is differentially steered, using two servo motors modified for continuous rotation. It is equipped with a Parallax BS2px24 microcontroller, capable of executing around $19,000$ BASIC instructions per second. The microcontroller is connected to a Parallax servo controller which, in turn, sends commands to the two servo motors. Two separate 6V power supplies are used: A set of four standard AA batteries give power to the microcontroller and the servo controller, whereas five large (D size), rechargeable batteries, mounted under the chassis as shown in the right panel of Fig. 8.1, provide power to the servos. The latter batteries are capable of delivering a full $8,000$ mAh of energy, allowing the robot to operate for several hours without recharging.

The sensory array (not yet mounted) will consist of IR proximity detectors, bumper switches, accelerometers, and a compass. In the full-scale prototype of the robot, the servo motors will probably be replaced by stepper motors capable of producing higher torque than the servo motors used in the first prototype. The full-scale version will also include the laser range finder from Hokuyo [26], as well as several other hardware components, such as a text-to-speech synthesizer, for robot-user communication.

Chapter 9

Conclusions and further work

In this chapter, a summary of the main conclusions are given, in very condensed form. In addition, a few directions for further work are provided.

9.1 Conclusions

As indicated by its title, the two main topics considered in this thesis have been (1) the generation of behaviors and (2) the organization of behaviors. The lists below briefly summarize the conclusions from the investigations presented in Papers I–X.

9.1.1 Generation of behaviors

- Energy efficient bipedal gaits can be achieved through evolutionary optimization of GFSMs in the case of a 2D five-link bipedal robot (Paper I).
- Smooth, human-like gait (cyclic in nature) can be achieved by guiding the evolutionary algorithm towards energy-minimal solutions (Paper I).
- Clusters of CPGs, connected through a feedback network, can be used for generating a bipedal gait in a simulated 3D bipedal robot by means of evolutionary optimization of a feedback network (Paper IV).
- Evolutionary optimization of RNNs can generate fairly good motor behaviors. However, their ability to generalize to changes in the initial conditions is limited (Paper III).

9.1.2 Organization of behaviors

The main conclusion from the many investigations involving the UF method is that the method, as implemented in the software library UFLIB, does indeed allow the user to set up and carry out the necessary simulations for a wide variety of (single-robot) behavioral organization problems, using a minimum of hand-coding (i.e. manual fine-tuning of parameters).

Further it may be concluded that:

- While the UF method is capable of solving the behavior selection problem, the performance of the resulting systems may be affected by the quality of the constituent behaviors (Paper III).
- If a polynomial ansatz is used for the utility functions in the UF method, a polynomial degree of three (or higher) should be chosen over a smaller degree, at least in cases involving fewer than six behaviors (Paper VI). However, as the search space rapidly increases with the polynomial degree, high polynomial degrees will result in longer optimization times.
- The mutation rate for the EA optimizing the utility functions should be set to a value that, on average, generates around three parameter modifications of an individual. If a lower mutation rate is used, the search for good solutions will generally be slower (Paper VI).
- Utility function polynomials including all terms up to a certain degree produce results that are equal to (or better than) polynomials with an arbitrary number of terms (Paper VII).
- The standard UF crossover operator, which swaps entire utility function polynomials between individuals, performs at least as well as operators capable of swapping individual polynomial terms (Paper VII).
- The use of multiple simulations (for the evaluation of a robot) generates significantly improved results, provided that the individual simulations are sufficiently long so that, in any given simulation, the robot is faced with the problem of making several difficult decisions concerning behavior activation (Paper VIII).
- The choice of a composite fitness measure (minimum, average, or a combination thereof) in the case of evaluations consisting of multiple simulations, does not have any significant effect on the ability of a robot to generalize to previously unseen situations (Paper VIII).

- The approach of providing a simulation *library*, from which new executable files, tailored to the problem at hand, easily can be generated, has proven to be very useful. Indeed, in most of the investigations concerning the UF method, the generation of the executable program was one of the *least* time-consuming procedures.
- The use of template-based, object-oriented text files for the specification of robots, arenas etc. greatly simplifies the procedure of setting up simulations of autonomous robots.

9.2 Further work

As most investigations performed in this thesis have been carried out in simulations, the next step should involve implementation in real robots. This is the aim of the transportation robot project described in Section 8.2 and outlined in Paper IX, where the UF method, and its implementation in UFLIB, will be further investigated and developed. An issue of particular importance will be to test the robustness of the UF method in progressively more complex real-world situations. Furthermore, in this project, at least one full-scale prototype will be constructed to allow, for example, tests of suitable sensor modalities for localization. The long-term vision is to generate a transportation robot that is able to carry out its tasks reliably in an arbitrary indoor environment.

In order to simplify further the use of the UF method, an issue that will become ever more important as the size of the behavioral repertoire increases far beyond a few behaviors, efforts will be made to automate tasks such as the specification of the behavioral hierarchy and the generation of definition files for simulations based on UFLIB.

Another interesting topic for further work would be to apply the UF method in connection with software agents used as a support in decision-making in, for example, risk management, insurance, finance etc. In this context, the concept of multiple simulations used in the UF method is also likely to become quite important, since the evolution of such agents will require exposure to many different situations.

Bibliography

[1] ACOSTA-MÁRQUEZ, C. A. and BRADLEY, D., "The analysis, design and implementation of a model of an exoskeleton to support mobility", in *Proceedings of the 2005 IEEE 9^{th} International Conference on Rehabilitation Robotics (ICORR 2005)*, pp. 99–102, 2005.

[2] AGV PRODUCTS. `http://www.agvp.com`.

[3] ARKIN, R. C., *Behavior-based robotics*. Cambridge, MA: The MIT Press, 1998.

[4] BERNOULLI, D., "Specimen theoriae novae de mensura sortis", *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, vol. 5, pp. 175–192, 1738.

[5] BLUMBERG, B. M., "Action-selection in Hamsterdam: Lessons from ethology", in *From Animals to Animats 3: Proceedings of the* 3^{rd} *International Conference on Simulation of Adaptive Behaviour (SAB94)*, MIT Press, 1994.

[6] BORLAND, "Delphi", `http://www.borland.com/delphi`.

[7] BROOKS, R., "A robust layered control system for a mobile robot", *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, 1986.

[8] BÄCK, T., FOGEL, D. B., and MICHALEWICZ, Z., *Handbook of evolutionary computation*. Bristol, UK: Institute of Physics, 1997.

[9] CHERNOFF, H. and MOSES, L., *Elementary decision theory*. New York, NY, USA: Dover Publications, Inc., 1986.

[10] COLLETT, T. and COLLETT, M., "Memory use in insect visual navigation", *Nature Reviews Neuroscience*, vol. 3, pp. 542–552, July 2002.

[11] COLLINS, S., RUINA, A., TEDRAKE, R., and WISSE, M., "Efficient bipedal robots based on passive dynamic walkers", *Science Magazine*, vol. 307, pp. 1082–1085, February 2005.

[12] DAWKINS, R., *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*. W. W. Norton, 1996.

[13] DAWKINS, R., *Climbing Mount Improbable*. W. W. Norton, 1997.

[14] DURRANT-WHYTE, H. F., "An autonomous guided vehicle for cargo handling applications", *International Journal of Robotics Research*, vol. 15, no. 5, pp. 407–440, 1996.

[15] FICICI, S., WATSON, R., and POLLACK, J., "Embodied evolution: A response to challenges in evolutionary robotics", in *Proceedings of the* 8^{th} *European Workshop on Learning Robots* (WYATT, J. L. and DEMIRIS, J., eds.), pp. 14–22, 1999.

[16] FLOREANO, D. and MONDADA, F., "Evolution of Homing Navigation in a Real Mobile Robot", *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, vol. 26, no. 3, pp. 396–407, 1996.

[17] FREEPASCAL. `http://www.freepascal.org`.

[18] FUJIMOTO, Y. and KAWAMURA, A., "Simulation of an autonomous biped walking robot including environmental force interaction", *IEEE Robotics and Automation Magazine*, vol. 5, no. 2, pp. 33–42, 1998.

[19] FURUSHO, J. and MASUBUCHI, M., "Control of a dynamical biped locomotion system for steady walking", *Journal of Dynamic Systems, Measurements, and Control*, vol. 108, pp. 111–118, 1986.

[20] GECKO SYSTEMS. `http://www.geckosystems.com`.

[21] GLSCENE. `http://glscene.org`.

[22] GOMEZ, F. and MIIKKULAINEN, R., "Incremental evolution of complex general behavior", *Adaptive Behavior*, vol. 5, pp. 317–342, 1997.

[23] GRILLNER, S., "Neural networks for vertebrate locomotion", *Scientific American*, pp. 48–53, January 1996.

[24] HART, P. E., NILSSON, N. J., and RAPHAEL, B., "A formal basis for the heuristic determination of minimum cost paths in graphs", *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-4, pp. 100–107, July 1968.

[25] HAYKIN, S., *Neural Networks: A comprehensive foundation*. Upper Saddle River, New Jersey: Prentice Hall, 2nd ed., 1999.

[26] HOKUYO. `http://www.hokuyo-aut.jp`.

[27] HOLLAND, J., *Adaptation in natural and artificial systems*. Cambridge, MA: MIT Press, 1992.

[28] IVANENKO, Y. P., POPPELE, R. E., and LACQUANITI, F., "Motor control programs and walking", *The Neuroscientist*, vol. 12, no. 4, pp. 339–348, 2006.

[29] JAKOBI, N., HUSBANDS, P., and HARVEY, I., "Noise and the reality gap: The use of simulation in evolutionary robotics", in *ECAL* (MORÁN, F., MORENO, A., GUERVÓS, J. J. M., and CHACÓN, P., eds.), vol. 929 of *Lecture Notes in Computer Science*, pp. 704–720, Springer, 1995.

[30] KANDEL, E. R., SCHWARTZ, J. H., and JESSELL, T. M., *Priniciples of Neural Science*. McGraw-Hill/Appleton & Lange, 4th ed., January 2000.

[31] KENNEDY, J. and EBERHART, R. C., "Particle swarm optimization", in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 4, pp. 1942–1948, 1995.

[32] KHATIB, O., "Real-time obstacle avoidance for manipulators and mobile robots", in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 500–505, 1985.

[33] KIRKPATRICK, S., GELATT, C. D., and VECCHI, M. P., "Optimization by simulated annealing", *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[34] KOZA, J. R., *Gentetic Programming: On the programming of Computers by Natural Selection*. Cambridge, MA: MIT Press, 1992.

[35] LUCIDARME, P. and LIÉGEOIS, A., "Learning reactive neurocontrollers using simulated annealing for mobile robots", in *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, pp. 674–679, 2003.

[36] MAES, P., "Modeling adaptive autonomous agents", *Artificial Life*, vol. 1, no. 1-2, pp. 135–162, 1994.

[37] MAES, P., "How to do the right thing", *Connection Science Journal*, vol. 1, no. 3, pp. 291–323, 1989.

[38] MATSUOKA, K., "Mechanisms of frequency and pattern control in the neural rhythm generators", *Biological Cybernetics*, vol. 56, pp. 345–353, July 1987.

[39] MCCLELLAND, J. L., RUMELHART, D. E., and HINTON, G. E., "The appeal of parallel distributed processing", in *Parallel Distributed Processing: Volume 1: Foundations* (RUMELHART, D. E., MCCLELLAND, J. L., and OTHERS, eds.), pp. 3–44, Cambridge: MIT Press, 1987.

[40] MCFARLAND, D., *Animal behavior: Psychobiology, Ethology and Evolution*. Longman, 3rd ed., 1999.

[41] MCFARLAND, D. and BÖSSER, T., *Intelligent behavior in animals and robots*. The MIT Press, 1993.

[42] MCGEER, T., "Passive dynamic walking", *International Journal of Robotics Research*, vol. 9, no. 2, pp. 68–82, 1990.

[43] MIGLINO, O., LUND, H. H., and NOLFI, S., "Evolving mobile robots in simulated and real environments", *Artificial Life*, vol. 2, no. 4, pp. 417–434, 1995.

[44] MITCHELL, M., *An introduction to genetic algorithms*. The MIT Press, 1996.

[45] MORAVEC, H., *Robot: Mere machine to transcendent mind*. Oxford University Press, 1999.

[46] MORIARTY, D. E. and MIIKKULAINEN, R., "Evolving obstacle avoidance in a robot arm", in *From Animals to Animats 4: Proceedings of the* 4^{th} *International Conference on Simulation of Adaptive Behavior (SAB'96)*, pp. 468–475, MIT Press, 1996.

[47] NIELSEN, J., "How we walk: Central control of muscle activity during human walking", *The Neuroscientist*, vol. 9, no. 3, pp. 195–204, 2003.

[48] NOLFI, S. and FLOREANO, D., *Evolutionary robotics*. The MIT Press, 2000.

[49] PIRJANIAN, P., "Behavior-coordination mechanisms – state-of-the-art", Technical report IRIS-99-375, Institute for Robotics and Intelligent Systems, University of Southern California, October 1999.

[50] PIRJANIAN, P. and MATARIC, M., "A decision-theoretic approach to fuzzy behavior coordination", in *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*, (Monterey, CA), November 1999.

[51] POLLACK, M. E. and OTHERS, "Pearl: A mobile robotic assistant for the elderly", in *AAAI Workshop on Automation as Caregiver*, August 2002.

[52] PUGH, J., ZHANG, Y., and MARTINOLI, A., "Particle swarm optimization for unsupervised robotic learning", in *Swarm Intelligence Symposium*, pp. 92–99, 2005.

[53] RECHENBERG, I., *Evolutionsstrategien*. Holtzmann-Froboog, 1994.

[54] RIEGL. `http://www.riegl.com`.

[55] ROSENBLATT, J., "DAMN: A distributed architecture for mobile navigation", *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 339–360, 1997.

[56] RUSSELL, S. J. and NORVIG, P., *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd ed., 2002.

[57] SANDHOLT, H., PETTERSSON, J., and WAHDE, M., "Development of a bipedal robot with genetic algorithm based motion control", in *Proceedings of the 8^{th} UK Mechatronics Forum International Conference (Mechatronics 2002)*, pp. 489–498, 2002.

[58] SASTRY, S. and BODSON, M., *Adaptive control: Stability, convergence, and robustness*. Prentice-Hall, 1989.

[59] SAVAGE, J., MARQUEZ, E., PETTERSSON, J., TRYGG, N., PETERSSON, A., and WAHDE, M., "Optimization of waypoint-guided potential field navigation using evolutionary algorithms", in *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*, pp. 3463–3468, 2004.

[60] SHEPHERD, G. M., *Neurobiology*. Oxford University Press, 3rd ed., May 1994.

[61] SHI, W. and ZUSMAN, D. R., "Fatal attraction", *Nature*, vol. 366, pp. 414–415, 1993.

[62] SIMMONS, R., GOODWIN, R., HAIGH, K. Z., KOENIG, S., and O'SULLIVAN, J., "A layered architecture for office delivery robots", in *Proceedings of the first international conference on Autonomous agents*, pp. 245–252, 1997.

[63] STADDON, J. E. R., *Adaptive dynamics: The theoretical analysis of behavior*. Cambridge, Massachusetts: The MIT Press, 2001.

[64] TUG (University of Maryland, `http://www.umm.edu/news/releases/robot.html`).

[65] VON NEUMANN, J. and MORGENSTERN, O., *Theory of Games and Economic Behavior*. Princeton, N. J.: Princeton University Press, 3rd ed., 1953.

[66] WAHDE, M., "A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions", *Journal of Systems and Control Engineering*, vol. 217, pp. 249–258, September 2003.

[67] WAHDE, M., "Evolutionary robotics: The use of artificial evolution in robotics", Tutorial presented at AMiRE2005, 2005. Available from: `http://www.me.chalmers.se/~mwahde/AdaptiveSystems/Tutorials.html`.

[68] WAHDE, M., *An introduction to adaptive algorithms and intelligent machines*. Göteborg: Chalmers Reproservice, 5th ed., 2006.

[69] WAHDE, M. and PETTERSSON, J., "UFLibrary demo", Available at: `http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary/Demo.html`.

[70] WAHDE, M. and PETTERSSON, J., "UFLibrary tutorial", Available at: `http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary`.

[71] WAHDE, M. and PETTERSSON, J., "A brief review of bipedal robotics research", in *Proceedings of the* 8^{th} *UK Mechatronics Forum International Conference (Mechatronics 2002)*, pp. 480–488, June 2002.

[72] WAHDE, M. and SANDHOLT, H., "Evolution of complex behaviors on autonomous robots", in *Proceedings of the* 7^{th} *UK Mechatronics Forum International Conference*, (Oxford), Pergamon Press, 2000.

[73] WATSON, R. A., FICICI, S. G., and POLLACK, J. B., "Embodied evolution: Embodying an evolutionary algorithm in a population of robots", in *Proceedings of the Congress on Evolutionary Computation*, vol. 1, pp. 335–342, IEEE Press, 1999.

[74] WATSON, R. A., HORNBY, G. S., and POLLACK, J. B., "Modeling building-block interdependency", in *Parallel Problem Solving from Nature – PPSN V*, (Berlin), pp. 97–106, 1998.

[75] WOLFF, K. and NORDIN, P., "Evolution of efficient gait with humanoids using visual feedback", in *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, (Tokyo, Japan), pp. 99–106, November 2001.

[76] YAO, X., "Evolving artificial neural networks", *Proceedings of the IEEE*, vol. 87, pp. 1423–1447, September 1999.

[77] ZEHR, E. P. and DUYSENS, J., "Regulation of arm and leg movement during human locomotion", *The Neuroscientist*, vol. 10, no. 4, pp. 347–361, 2004.

Paper I

A flexible evolutionary method for the generation and implementation of behaviors for humanoid robots

in

Proceedings of the IEEE-RAS International Conference on Humanoid Robots (Humanoids 2001), Tokyo, Japan, November 2001, pp. 279–286.

A flexible evolutionary method for the generation and implementation of behaviors for humanoid robots

Jimmy Pettersson, Hans Sandholt, Mattias Wahde

Division of Mechatronics, Chalmers University of Technology,
412 96 Göteborg, Sweden
{jimmy.pettersson, hans.sandholt, mattias.wahde}@me.chalmers.se

Abstract

A flexible method for generating behaviors for bipedal robots is presented and applied to the case of motor behaviors. The method is biologically inspired and is based on evolutionary algorithms in connection with generalized finite state machines (FSMs). The evolutionary process acts directly on the FSMs and optimizes both their parameters and their structure.

In this method, only a rough indication of the desired behavior needs to be specified as an initial condition to the evolutionary algorithm, which then performs further optimization of the behavior.

We apply the method to two test cases, namely energy optimization and robust balancing. It is found that the method performs very well in both cases, and that its ability to modify the structure of the FSMs is very useful. In the case of energy optimization, the walking length for a given amount of energy is improved by 134 %.

Keywords: bipedal robots, evolutionary robotics, behavior–based robotics

1. Introduction

During the early decades of the 21st century, it is expected that humanoid robots will come to play an increasingly important role, both in industries and as household robots. However, in order for this to happen, the robots will need to become much more complex than today, and the development of such robots presents a formidable challenge to researchers and engineers. As the complexity of humanoid robots increases, there will be a strong need for a flexible and versatile representation for motor behaviors (and other behaviors) [9]. In addition to a flexible representation, an efficient optimization method for generating robust and energy-optimal motor behaviors will also be needed.

The development of a representation and the choice of an optimization method are difficult problems. However, the fact that the systems that are being generated – humanoid robots – are modelled on biological systems – humans – indicates that it would be wise to consider optimization methods inspired by biological considerations, such as e.g. evolutionary algorithms.

The application of evolutionary computation to robotics has given rise to the very active research field of evolutionary robotics [12]. The use of evolutionary methods to the case of bipedal robots has mainly been restricted to parameteric optimization within a pre-specified structure (see e.g. [1], [3], [4], and [6]). Notable exceptions are provided by Arakawa and Fukuda [1], who allowed a certain flexibility in the representation of the control system and Paul and Bongard [13], who allowed the morphology of the bipedal robot to vary.

The aim of this paper is to introduce a flexible and general method for the construction of robotic behaviors. We will describe the representation of the behaviors, and also show how evolutionary optimization can be applied successfully to this representation, optimizing not only the parameters of the system but also its structure. While the focus of the paper is on the description of the method as such, we will also present some early results obtained with this method.

2. The robot

For our simulations, we have used a five-link robot, constrained to move in the sagittal plane. The robot has five degrees of freedom: torques can be applied at both knee joints and at both hip joints. In addition, a fifth actuator controls the posture of the upper body. The lengths of the leg links have been based on the corresponding values for a 1.5 m tall human.

The structure of our robot, which is shown in Fig. 1 is similar to that earlier used by e.g. Cheng and Lin [3] and Mitobe *et al.* [10]. While this robot model is perhaps somewhat simplistic, it is still sufficient

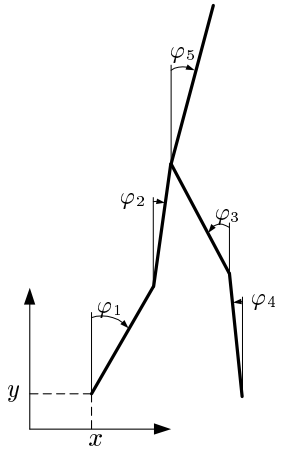


Figure 1: Configuration of the bipedal walking robot.

for the purposes of demonstrating the feasibility of our method for representing behaviors for bipedal robots. We have used a lagrangian formulation for the equations of motion (see e.g. [11], Ch.4), which take the form

$$\mathbf{M}(\mathbf{z})\ddot{\mathbf{z}} + \mathbf{C}(\mathbf{z},\dot{\mathbf{z}})\dot{\mathbf{z}} + \mathbf{N}(\mathbf{z}) + \mathbf{A}^{\mathrm{T}}\boldsymbol{\lambda} = \boldsymbol{\Gamma}, \quad (1)$$

where $\mathbf{M}$ is the generalized inertia matrix, $\mathbf{C}$ contains centrifugal and Coriolis terms, $\mathbf{N}$ contains gravity terms, $\mathbf{A}$ is the constraint matrix and $\boldsymbol{\lambda}$ the corresponding Lagrange multipliers, and $\boldsymbol{\Gamma}$ contains the generalized forces. The derivation of the various matrices and vectors is straightforward, and thus will not be given here. The generalized coordinate vector $\mathbf{z}$ is given by

$$\mathbf{z} = [\varphi_1, \ldots, \varphi_5, x, y]^{\mathrm{T}}, \quad (2)$$

where the angular variables $\varphi_1, \ldots, \varphi_5$ determine the orientation of the limbs (see Fig. 1), and x, y are the coordinates for one foot (i.e. the tip of a leg) of the robot.

The vector of generalized forces $\boldsymbol{\Gamma}$ is related to the torques $\mathbf{T}$ applied at the five joints through the transformation $\boldsymbol{\Gamma} = \mathbf{D}\mathbf{T}$, where

$$\mathbf{D} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (3)$$

The constraint matrix $\mathbf{A}$ varies in size and structure depending on the number of feet (0, 1, or 2) that are in contact with the ground [7].

Lagrange's equation for impulsive motion is used to model ground impacts and perturbations and is stated as

$$\left.\frac{\partial \mathrm{T}}{\partial \dot{\mathbf{z}}}\right|_{t^+} - \left.\frac{\partial \mathrm{T}}{\partial \dot{\mathbf{z}}}\right|_{t^-} = \hat{\mathbf{Q}}\ , \quad (4)$$

where t^+ and t^- denote the instants immediately after and immediately before the impulse, respectively, $\hat{\mathbf{Q}}$ is the vector of generalized impulses, and T is the kinetic energy of the system. Using the fact that the generalized inertia matrix ($\mathbf{M}$) is symmetric, the generalized momenta can be expressed as: $\partial \mathrm{T}/\partial \dot{\mathbf{z}} = \mathbf{M}\dot{\mathbf{z}}$, which, when inserted into Eq. (4), gives the generalized postimpact velocities as

$$\dot{\mathbf{z}}^+ = \mathbf{M}^{-1}\hat{\mathbf{Q}} + \dot{\mathbf{z}}^-\ . \quad (5)$$

3. The method

The implementation of motor behaviors (and other behaviors) in robots consists of two parts which will now be introduced: an architecture for storing the behaviors of the robot, and a method for obtaining the behaviors that are to be implemented.

3.1 The representation

While this paper will deal exclusively with bipedal *motor* behaviors, the ultimate goal of this work is to arrive at a method which is sufficiently general to be able to accomodate not only bipedal gaits but also other aspects of the behavior of a robot[1], such as the ability to avoid obstacles, grip objects etc. Thus, an architecture which can *only* hold fully specified reference trajectories for bipedal gaits will not be sufficient.

Instead, we have chosen to use an architecture based on (generalized) finite state machines (FSMs). FSMs have the advantage of allowing combination of several behaviors into a complete behavioral repertoire [14], and they have often been used in connection with behavior-based robotics [2]. Furthermore, a system based on FSMs is generally transparent and easy to interpret.

A standard FSM consists, as the name implies, of a finite number of states and conditional transitions between those states. Furthermore, the allowed set of actions is usually chosen from a finite alphabet. The FSMs introduced in this paper are slightly different. First, each state in an FSM is here associated with a set of variables specific to that state, whereas in a standard FSM, the variables are associated with the transitions between states. In addition, we use

[1]For this reason, we will use the term *robotic brain* for the computer program that determines the actions of the robot, rather than the term *control system*. The latter term would indicate a more limited representation employing classical control theory.

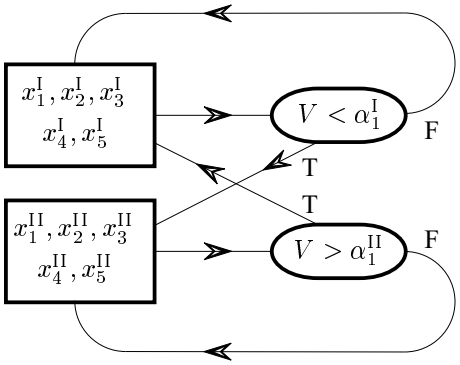


Figure 2: A simple two-state FSM, with five state variables and one transition condition per state. The arrows indicate the direction of signal flow. If the condition under consideration is true, the corresponding arrow marked with a T is followed. If instead the condition is false, the arrow marked with an F is followed.

continuous variables rather than a discrete alphabet. Each state has a number of conditional transitions, each with a specified target state.

A simple, generic, example of a two-state FSM is shown in Fig. 2. In this FSM both states contain the values of five variables (which may, for example, represent the reference angles for a given posture for the five–link bipedal robot). From the first state, the FSM can jump to the second state if the condition $V < \alpha_1^{\mathrm{I}}$ is fulfilled. Note that the variables V (of which only one was introduced in Fig. 2) defining the transition conditions need not be the same as the variables x_i^s specified in the states s. In this case, the condition variable V may, for instance, measure the deviation between the actual posture of the robot, and the posture specified in the active state. If the deviation is sufficiently small, the robot may proceed to the second state etc.

If no condition is fulfilled, the FSM remains in the same state, as indicated in Fig. 2 by the links emanating on the right hand side of the transition conditions. Note that, in subsequent figures, these links are not explicitly shown.

The number of transition conditions, as well as the number of variables defining the conditions, may vary from state to state. In cases where there is more than one transition condition associated with a state, the conditions are checked in order from left to right, so that the leftmost condition has the highest priority, since it is always checked.

3.2 The evolutionary algorithm

Evolutionary algorithms constitute, in our opinion, a natural choice for the generation of motor behaviors and other behaviors for autonomous robots in general, and bipedal robots in particular. After all, it is known that evolution is capable of generating highly complex structures in nature, and that evolutionary algorithms, which are based on natural evolution, often prove to be highly efficient in problems involving large and complicated search spaces. Clearly, the construction of robotic motor behaviors, which is the subject of this paper, is indeed a problem involving a very large search space.

The most commonly used type of evolutionary algorithm is the genetic algorithm (GA) [8]. Most of the work to date on evolutionary algorithms in connection with bipedal robots has been based on GAs ([1], [3], [4], and [6]). However, standard genetic algorithms may not the best choice from the point of view of the construction of robotic brains. A standard GA is useful when carrying out parametric optimization, where the parameters of the system under study easily can be coded into a string of digits.

However, we wish to go beyond parametric optimization, and optimize not only the parameters but also the *structure* of the robotic brain. Thus, a more flexible scheme is required. The use of evolutionary algorithms in connection with FSMs, known as evolutionary programming, was pioneered by Fogel (see e.g. [5]). In evolutionary programming, the evolutionary process acts directly on the FSMs, by optimizing both the parameters of the FSMs and their structure, e.g. the number of states and transition conditions.

Our method is an adaptation of evolutionary programming to the case of generalized FSMs as described above, and it includes both crossover and mutation operators, by contrast with the original form of evolutionary programming which only used mutation operators.

Briefly, the process operates as follows: A fitness measure is specified before the simulation. An example of a fitness measure suitable for bipedal locomotion is given by the distance covered by the robot as it uses up a pre-specified amount of energy. In the beginning of a simulation, a population of random FSMs is generated. Normally, the initial population consists of rather simple FSMs. Then, all individuals in the population are evaluated, and each individual obtains a fitness value based on its performance.

The following sequence is then repeated until a satisfactory solution has been found: two individuals are selected from the population using tournament selection. Then, two offspring are formed by the procedures of crossover and mutation outlined below. The two new individuals are then inserted into the population, replacing the two worst individuals. Finally the two new individuals are evaluated, and the procedure is repeated again.

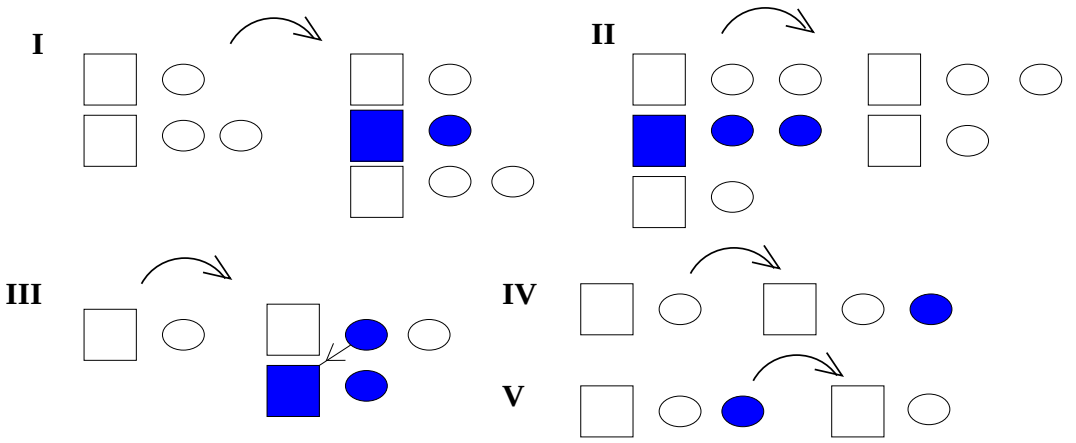


Figure 3: Structural mutations: I) *Insert state*: inserts a state with one transition condition, whose variables are defined as the average of the variables in the two adjacent states, II) *Delete state*: simply removes a state, III) *Add prioritized state*: adds, to an already present state, a transition (with top priority) to a new state. The variables of the new state are taken as slight mutations of the variables in the state to which the new transition was added, IV) *Add transition condition*: adds a transition condition (with lowest priority) to a state, and, V) *Delete transition condition*: deletes the transition condition with lowest priority for a given state. Note that, for clarity, the transitions are not explicitly shown (except one transition in case III) in this figure.

3.2.1 Crossover

Combination of material from different individuals is an important part of evolutionary algorithms. Crossover is easy to implement in a standard GA, but somewhat more difficult in our case, in which the structures to be crossed are more complicated than the strings used in GAs. We have chosen to introduce a crossover procedure which simply swaps two selected states between two FSMs. The procedure begins by the selection of one state in each of the FSMs that are to be crossed. Next, the states with their transition conditions are swapped between the FSMs, forming two new FSMs. As a final step, it is checked that the targets for the conditional jumps are consistent, i.e. that no condition generates a jump to a non-existent state (which may occur if the FSMs contain different numbers of states). If an inconsistent jump is detected, the target is arbitrarily set to state 1. This does not imply a significant restriction, since subsequent mutations can change the transition target to any of the available states.

3.2.2 Mutations

Two kinds of mutations are used: *parametric mutations*, which modify the value of any parameter in the FSM by a small, random amount, and *structural mutations* which modify the structure of the FSMs. The structural mutations, which are needed in order to arrive at the desired flexibility, are illustrated in Fig. 3.

3.3 The simulation program

The generalized FSM representation and the evolutionary algorithm described above have been implemented in a computer program written in Delphi Object-oriented Pascal. The program is fully object-oriented, so that the data structures, e.g. the FSMs, are flexible and can be of arbitrary size and complexity. Thus, the program permits an open-ended evolutionary process that can lead to very complex structures.

At the outset of a simulation, the user provides a set of parameters, such as link lengths and masses (for the robot), the fitness measure, initial structural parameters for the FSMs (e.g. the number of states) as well as ranges for the parameters (variables and transition conditions) defining the states. Parameters related to the simulation of a single individual, such as e.g. the length of the time steps for the numerical integration of the equations of motion, must also be specified. Furthermore, it is possible to provide limits on the joint torques and their first derivative with respect to time.

The user may also choose between two different types of initial FSMs, *linear FSMs*, in which each state s has a single transition condition whose target is state $s + 1$, except for the last state, for which the target of the transition condition is state 1, and *general FSMs*, with a completely arbitrary structure. The linear FSMs are useful for generating cyclic behaviors, such as a step sequence, whereas the more flexible general FSMs are needed e.g. to cope with perturbations during a step or other non-cyclic motor behaviors. Note that the specification of an FSM

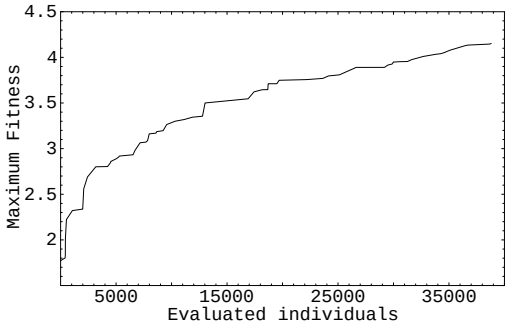


Figure 4: Fitness of the best individual as a function of the number of individuals for test case 1 (energy optimization).

type only relates to the *initial* population. The evolutionary process has full freedom to add and delete states, as outlined above, should the need arise.

In keeping with the aim of developing a sufficiently flexible representation that can hold different kinds of behaviors, great care has been taken to make the data structures for the FSMs as general as possible. Thus, an FSM can consist of states of many different types (i.e. with different variables defining the states), and with various transition conditions of, in principle, any form.

However, here we are concerned with motor behaviors, and we have therefore used a specific kind of FSM, the components of which will now briefly be described.

FSM states In any state of the FSMs used here, the requested torque at joint i is given by

$$\tau_i^{\text{req}} = K_i^P(\varphi_i - \varphi_i^{\text{ref}}) + K_i^D\dot{\varphi}_i + K_i^0 \quad (6)$$

where K_i^P, φ_i^{ref}, K_i^D, and K_i^0 are constants. Thus, for the representation of motor behaviors, each FSM state holds a set of 20 variables (4 for each link). Since we, for realism, normally impose limits on the torque derivatives, the actual torque delivered at a joint is not always equal to the requested torque. In most situations, however, the actual torque approaches the requested torque within a few time steps.

Transition conditions For each state s, there are N^s transition conditions which, in this case, take the form

$$\text{if } (V_i\,[\text{Op}]\,\alpha_k) \text{ then jump to target}, \quad (7)$$

where [Op] denotes one of the operators $<$ and $>$, α_k is a constant, specific to transition condition k, and the target is any state in the FSM (cf. Fig. 2).

	Early FSM	Best FSM
Energy used (J)	500	500
Length walked (m)	1.77	4.15
Total time (s)	2.56	4.11
Average speed (m/s)	0.69	1.01

Table 1: A comparison between the first individual that managed to walk (left) and the best individual, in test case 1.

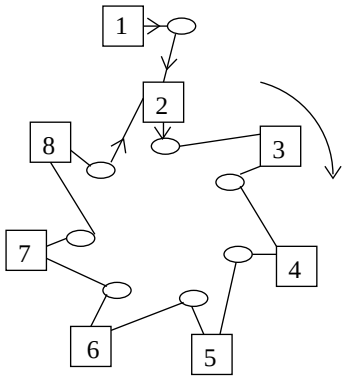


Figure 5: Structure of the best FSM obtained in test case 1 (energy optimization).

The variables V_i can be choosen freely. In this application, we have chosen to use six condition variables, namely

$$V_i = \varphi_i - \varphi_i^{\text{ref}},\ i = 1, \ldots, 5, \quad (8)$$

and

$$V_6 = \sqrt{\frac{1}{5}\sum_{i=1}^{5}(\varphi_i - \varphi_i^{\text{ref}})^2}. \quad (9)$$

4. Results

In order to test the efficiency of both the representation and the evolutionary algorithm, a number of runs of the simulation program have been made. Two specific applications have been used as test cases, namely the generation of smooth and energy-efficient bipedal gaits, and the construction of robust balancing in the presence of perturbations.

4.1 Test case 1: Energy optimization

For any autonomous robot that carries its own energy source (e.g. batteries), it is clearly of paramount importance to move with as little use of energy as possible. In nature, evolution has optimized human walking (and, in general, animal locomotion), to make it very energy efficient. While we do apply

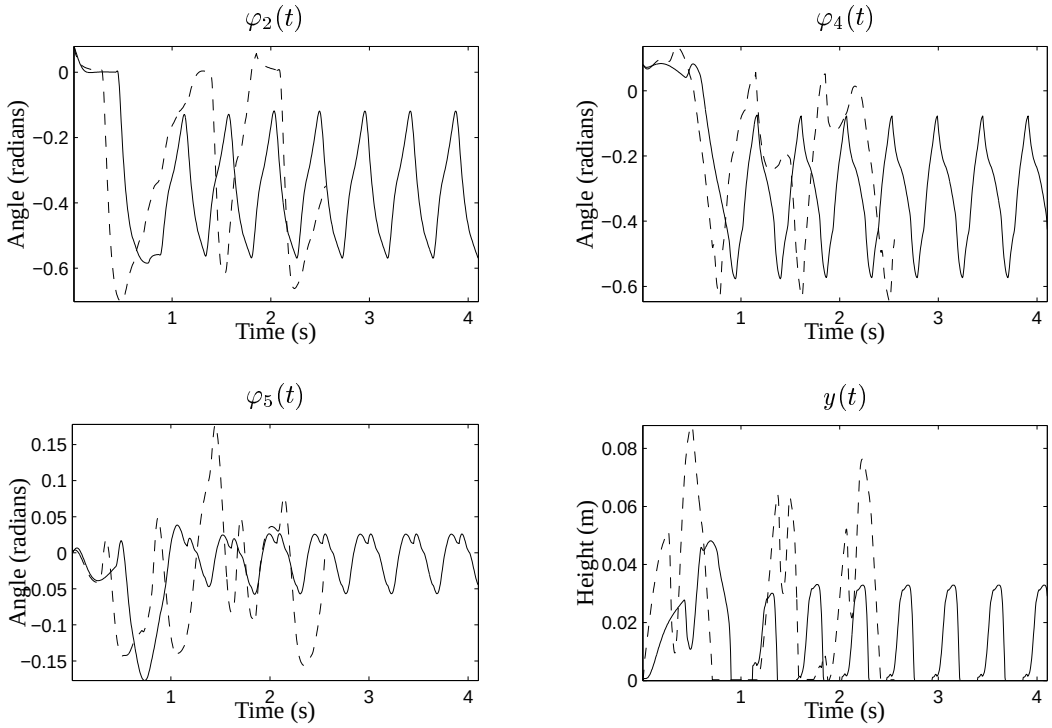


Figure 6: Energy optimization. Each plot shows the variation with time of one generalized coordinate for the first FSM that was able to make the robot walk (dashed) and the best FSM in the run (solid). Only some of the 7 generalized coordinates are shown in this figure.

artificial evolution to optimize the gait of our simulated robot, it should be pointed out that our optimization problem differs from the optimization carried out by natural evolution. In our case, the configuration of the robot, i.e. its bipedal nature and its structure with five links of given length and mass, are given whereas in natural optimization both the structure of the animal and its method of locomotion are optimized. However we do, as described above, allow a considerable freedom concerning the structure of the *brain* of the robot.

For the energy optimization runs, the fitness measure was chosen as the length walked by the robot until it had used an energy of 500 J. By using this fitness measure, energy optimization is obtained without explicitly having to include the energy usage in the fitness measure in an ad hoc fashion. In order to prevent the robot from walking very slowly, a time limit of 6 simulated seconds was introduced as well.

The population size was set to 400, and the structural and parametric mutation rates were set to 0.02 and 0.03, respectively. The crossover probability was equal to 0.10. The time step length was 0.005 seconds. Furthermore, limits were set on the maximum torque delivered at the joints (200 Nm), as well as the maximum rate of change of the torques (3000 Nm/s).

One of the main purposes with our method is to allow for the possibility of specifying, in a very loose sense, a sequence of motions, which will then be further optimized by evolution. In the development of energy-optimized gaits, we therefore specified only 8 reference states, 4 for the step with the left foot, and 4 for the right step.

The reference angles were set so as to generate a very rough representation of the two steps. The proportional and derivative constants were given random values centered on -250 Nm for the proportional constants and -15 Nms for the derivative constants. The K_i^0 parameters were given random values in the range $[-10, 10]$ Nm. The initial population consisted of linear FSMs (see Sect. 3.3).

In the beginning of the run, it was clear that the initial specification of the motion was much to rough to generate smooth walking: the few robots that managed to walk at all, stumbled forward in a very inefficient manner. Many robots used up their 500 J without getting anywhere. However, the optimization algorithm very quickly began to improve the gait, and the length walked by the robot increased considerably, from 1.77 m early in the run, to 4.15 m at the end, as shown in Fig. 4 and Table 1.

The total number of states of the best FSM at the end of the run was also equal to 8. However, this was in no way enforced. Indeed, during the run, several of the best FSMs that appeared used more than 8

states. The 8 states of the final FSM were totally different from the states specified in the beginning of the run.

Furthermore, the evolutionary optimization method was able to improve the structure of the FSM. Clearly, a cyclic sequence of states is convenient when walking at full speed. However, the robot starts from rest, and thus the very first part of the motion differs from the rest. This was indeed exploited: the structure of the best FSM at the end of the run contained one state that was used only to get the robot started, and 7 states that were used in a cyclic fashion for the continued motion, as shown in Fig. 5.

Finally, we note that the bipedal gait generated by the best FSM in the run was very smooth (see Fig. 6) and symmetric compared to the FSMs obtained early in the run, despite the fact that symmetry was not explicitly required.

4.2 Test case 2: Robustness

A bipedal robot moving in an unstructured environment, such as e.g. a busy street or a hospital, will invariably find itself in situations where it cannot rely on prespecified reference trajectories. For example, the robot may encounter an unexpected moving obstacle, or it may lose its balance due to an external perturbation or simply a bump in the ground. Thus, for such robots to be useful, they must be able to cope with unexpected situations. As a simple example, and as a test of our method, we have considered the following case: Assume that a bipedal robot is about to begin climbing some stairs, and as it lifts the front leg, it is perturbed. A sequence of three point perturbations, modeled as impulsive forces, are applied. The generalized velocities after each perturbation are computed using Eq. (5). The first perturbation is applied on the thigh of the supporting leg, the second on the upper body, and the third on the lower part of the lifted leg, as shown in the right panel of Fig. 7.

At the start of each simulation, the robot was placed with both feet on the ground, and the FSM of the robot contained a single state which made it lift the front leg. The fitness measure was defined simply as the inverse of the integrated total deviation between a desired position, with one leg lifted as shown in Fig. 7, and the actual position of the robot. The total deviation was computed as the root mean square of the deviation of each generalized coordinate. The fitness computation began after 0.6 s, giving the robot some time to reach the desired position from its starting position. Each simulation lasted for the equivalent of 3.6 s, and the three perturbations were applied after 0.8 s (perturbation a, see Fig. 7), 1.4 s (b), and 2.0 s (c), respectively. The

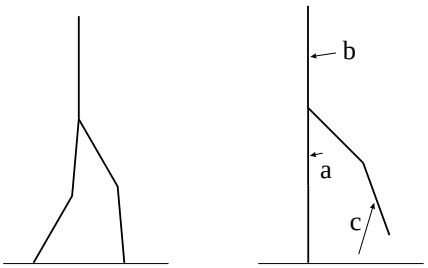


Figure 7: Starting posture (left) and desired posture for the robot in test case 2. The arrows indicate the magnitudes, directions, and points of application of the perturbations.

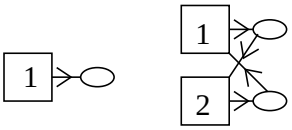


Figure 8: Initial (left) and final structure of the FSMs from test case 2. The added state helps the robot cope with the perturbations.

simulated robots were given a maximum of 500 J of energy to lift the leg and to handle the perturbations. The parameters of the evolutionary algorithm were the same as in test case 1 (see Sect. 4.1).

While the initial FSMs generally had severe difficulties in keeping the robot upright, FSMs capable of doing so appeared fairly quickly as a result of the optimization. More interestingly, the final FSM obtained from this run had undergone a structural mutation in which an additional state was added to cope with the perturbation. A schematic view of the structure of the initial FSMs and the best FSM obtained is shown in Fig. 8.

5. Discussion and Conclusion

In this paper, we have introduced a method for the generation of motor behaviors in bipedal robots. With our procedure, it is sufficient to provide the optimization algorithm with a rough indication of the desired motor behavior (rather than a complete trajectory specification), and then allow the algorithm to optimize it.

Ideally, it should be possible to generate a bipedal gait, or some other motor behavior, without specifying even a rough set of reference values. However, if no specification is made at all, it is not evident that a human-like gait will result. For instance, the evolutionary process may select a bird-like gait instead. Thus, some guidance should be given to the optimization algorithm, for instance in the form of a few reference positions as in our method.

It is obvious that for bipedal robots to be useful, they must be able to cope with unstructured and unpredictable environments. Our procedure may be useful in the construction of such robots, chiefly because of the structural flexibility of the corresponding robotic brains and the fact that the optimization method proceeds with a minimum of bias.

We believe that the ability to optimize the structure of the robotic brain, in addition to its parameters, is of great importance, and allows a kind of open-ended evolutionary process, which can produce structures that are much more complex than those initially specified. A possible indication supporting this hypothesis is the fact that the fitness values continued to increase during the full extent of the runs, rather than reaching a plateau quickly, as is often the case in evolutionary algorithms. A stronger indication is derived from the fact that the possibility to modify the structure of the FSMs was exploited in both of the test cases considered here. Thus, even though it probably would be possible, at least for simple gaits, to specify a useful FSM by other means (or even by hand), it has been our policy to give the evolutionary optimization method as much freedom as possible.

The two test cases also showed that considerable improvements could be obtained in a reasonable amount of time. In the case of energy optimization, a 134% improvement in walking length was obtained in a run that lasted approximately 28 hours on an 800 MHz pentium III computer.

The results presented here are, to a great extent, preliminary, and further experiments are underway to test the procedure in more challenging situations. The aim is to develop a full behavioral repertoire for bipedal locomotion using the procedure described in this paper, and to combine these behaviors using e.g. the method for evolutionary combination of separate behaviors described by Wahde and Sandholt [14]. Furthermore, we plan to implement the resulting robotic behaviors in the bipedal robot which is currently under development in our group.

References

[1] T. Arakawa and T. Fukuda, Natural Motion Trajectory Generation of Biped Locomotion Robot using Genetic Algorithm through Energy Optimization. In: *Proc. of the 1996 IEEE International Conference on Systems, Man and Cybernetics*, pp. 1495-1500, 1996

[2] R.C. Arkin, *Behavior-based robotics*, MIT Press, Cambridge, MA, 1998

[3] M.-Y. Cheng and C.-S. Lin, Genetic Algorithm for Control Design of Biped Locomotion. In: *Proc. of the 1995 IEEE International Conference on Systems, Man and Cybernetics*, pp. 1315-1320, 1995

[4] S.-H. Choi, Y.-H. Choi, and J.-G. Kim, Optimal Walking Trajectory Generation for a Biped Robot Using Genetic Algorithm. In: *Proc. of the 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1456-1461, 1999

[5] L. Fogel, *Intelligence through simulated evolution*, Wiley, NY, 1999

[6] T. Fukuda, Y. Komata, and T. Arakawa, Stabilization Control of Biped Locomotion Robot based Learning with GAs having Self-adaptive Mutation and Recurrent Neural Networks. In: *Proc. of the 1997 IEEE International Conference on Robotics and Automation*, pp. 217-222, 1997

[7] J. Furusho et al., Realization of Bounce Gait in a Quadruped Robot with Articular-Joint-Type Legs. In: In: *Proc. of the 1995 IEEE International Conference on Robotics and Automation*, pp. 697-702, 1995

[8] J.H. Holland, *Adaptation in Natural and Artificial Systems*, 1st ed. University of Michigan Press, Ann Arbor; 2nd ed. MIT Press, Cambridge, MA, 1992

[9] F. Kanehiro et al., Developmental Methodology for Building Whole Body Humanoid System. In: *Proc. of the 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1210-1215, 1999

[10] K. Mitobe et al., Non-linear feedback control of a biped walking robot. In: *Proc. of the 1995 IEEE International Conference on Robotics and Automation*, pp. 2865-2870, 1995

[11] R.M. Murray, Z. Li, and S.S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, CRC Press, 1994

[12] S. Nolfi and D. Floreano, *Evolutionary Robotics*, MIT Press, Cambridge, MA, 2000

[13] C. Paul and J.C. Bongard, The Road Less Travelled: Morphology in the Optimization of Biped Robot Locomotion. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2001)*, in press

[14] M. Wahde and H. Sandholt, Evolution of complex behaviors on autonomous robots. In: *Proc. of Mechatronics 2000, the 7th UK Mechatronics Forum International Conference*, Elsevier, 2000

Paper II

EvoDyn: A simulation library for behavior-based robotics

Technical report, Chalmers University of Technology, September 2003.

EvoDyn: A Simulation Library for Behavior-based Robotics

Jimmy Pettersson
Div. of Mechatronics
Dept. of Machine and Vehicle Systems
Chalmers University of Technology
412 96 Göteborg, Sweden
E-mail: jimmy.pettersson@me.chalmers.se

September 2003

Contents

1 Introduction

This paper describes a simulation library for evolutionary generation and optimization of behaviors for rigid-body-chain-type robots, such as e.g. bipedal robots. The core of the simulation library includes a rigid-body dynamics engine, based on Featherstone's algorithm as described in [11]. Developed in Delphi (object-oriented Pascal), it supports tree-structured rigid body systems described in terms of MDH parameters [4]. In addition, the library contains units for implementing robotic behaviors, as well as an evolutionary algorithm (EA) for optimization of such behaviors.

There exists many free simulators for rigid-body systems; examples are: (1) ODE, a dynamics engine for real-time simulation of articulated rigid body structures based on a Lagrange multiplier method for deriving the equations of motion; (2) BREVE, a simulation environment for decentralized systems and artificial life; (3) AERO, a library based on a penalty method (springs and dampers) and best suited for use in computer graphics; (4) DynaMechs, a library using the Featherstone algorithm for solving the equations of motion and suitable for the simulation of tree-structured systems; (5) DANCE, a software package for dynamic animation and control, based on (but not bounded to) the commercial package SD/FAST; (6) ABDULA, a C++ software library for physical animation of articulated solids. In addition to these free software packages, there are numerous commercially available packages such as MathEngine, Vortex, Dymola, Adams, Havok, and many more.

Most of the simulation packages mentioned above are focused on the solid-body dynamics. This is also an important part of the package presented here and, in particular, the minimization of computation times is the primary goal in the implementation of the dynamics engine. The main difference between this package and previous packages is that it includes units for robotic behaviors and for evolutionary optimization of such behaviors. Implementation of the evolutionary optimization is focused on *open-ended* problems, that is, including not only parametric optimization but also structural optimization while allowing the fitness measure to vary. Since optimal structures are not easily assigned in advance, structural optimization is an important part in the evolutionary process. In an EA that allows structural modifications, the problem of efficient simulations is also somewhat challenging from a programming point of view. The simulation library described here is aimed at simplifying the implementation of evolutionary optimization of robotic behaviors.

The next step in the continuing development of the simulation library will be to include a general method also for the evolutionary organization of behaviors but this has not yet been fully developed yet and is thus only briefly described here.

The report is organized as follows; first a very brief introduction to behavior-based robotics, robotic behaviors, and behavioral organization will be given, followed by a short introduction to the special case of bipedal walking robots. Next, the actual simulator will be described in some detail, starting with the solid body dynamics and continuing with the evolutionary optimization of behaviors.

2 Behavior-based robotics

The field of behavior-based robotics (BBR), which is developing at a rapid rate, is mainly concerned with autonomous robots, which are supposed to function in unstructured environments, where the robot often finds itself in situations not previously encountered [1].

In BBR, there is a strong coupling between perception and reaction, where sensors are the main source of information. Unlike classical artificial intelligence (AI), BBR does not use explicit internal world models. Building such models in which, for example, reference trajectories are constructed, is computationally expensive and reduces the reactiveness of a robot.

Behavior-based robotics uses a bottom-up approach for the generation of robotic brains (control systems), where complex behaviors are built by combining several simple behaviors. Examples of such simple behaviors are; *find energy source*, *avoid obstacles*, *pursue object* etc. Behaviors are often inspired by animal behaviors as observed by ethologists [10], and are commonly generated by means of biologically inspired computation methods.

Behaviors can be implemented using a variety of methods, ranging from classical control (e.g. PD controllers) for low-level motor behaviors to artificial neural networks (ANNs) and finite state machines (FSMs) for complex motor behaviors and non-motor (e.g. cognitive) behaviors.

Unlike industrial robots (manipulators), where the accuracy of the end-effector is crucial for the applicability of the robot, autonomous robots do not require the same amount of precision. Hence, the use of more flexible and adaptive methods for behavior generation and implementation, such as biologically inspired methods, becomes a suitable choice.

One of the main difficulties in BBR is the organization of behaviors, i.e. how the selection of different behaviors should be made at any given instant. Given information from relatively simple and incomplete sensors, choosing the appropriate behavior is often extremely difficult. Many methods rely on hand-coded settings. While useful in simple cases, such methods often lead to solutions that are not robust [19] in more complex cases involving several behaviors. In this case, the connection to biology is important, since even the simplest of animals are able to make intelligent choices (behavioral selection) within their environments.

Since behaviors constitute the building blocks in BBR, solving the problem of behavioral organization is crucial for the development of the field. It should be noted that the research field is still very young, around 20 years, and is constantly developing. Up until now, seemingly simple problems have been investigated, and the current level of intelligence in autonomous robots is considered to be at the level of insects [13].

2.1 Robotic behaviors

In behavior-based robotics, a behavior is a unit that produces an action based on sensory information, and is responsible for handling a part of the robot's overall task. In the current version of the simulator presented here, focus is mainly on robotic motor behaviors, i.e. behaviors that produce torques or forces used for moving a robot or a part of it.

For robots that are intended to move in controlled environments, such as e.g. a work cell in a factory, motor behaviors based solely on pre-defined reference trajectories are suitable. However, autonomous robots require the use of more adaptive systems in order to function in unstructured environments full of uncertainties.

Examples of representations that are useful in the implementation of such systems are ANNs, generalized finite state machines (GFSMs), and fuzzy logic controllers (FLCs).

In contrast to ANNs, both GFSMs and FLCs are representations which are easy to interpret. When using biologically inspired computation methods, such as evolutionary algorithms, interpretability of the final result might be an important factor, and even more so in the case of open-ended problems, where not only the parameters are modified but also the structure. On the other hand, ANNs have several other useful features, such as graceful degradation in the case of neuron loss and the ability to generalize to situations not previously encountered.

2.2 Behavioral generation

Basic behaviors, such as *avoid obstacle*, *find energy source*, *follow target*, etc. can be generated in a number of different ways. The simplest types of behaviors can be implemented by hand, whereas more complex types of behaviors can be generated by means of, for instance, an **evolutionary algorithm**. Other methods for behavior generation includes **reinforcement learning**, which is based on rewards and punishments, **neural networks**, where learning is achieved through changes in the synaptic weights, and **fuzzy control**, which is a rule-based system (see [1] and references therein for more detailed information).

2.3 Behavioral organization

Assuming that there exists a repertoire of behaviors; how to select the most appropriate behavior at any given instant is a difficult problem.

For obvious reasons, the importance of a certain behavior will vary with time. The robotic brain must, at any given instant in time, choose the most suitable behavior in order to perform the task it was designed for.

Several methods have been proposed for behavioral organization, including the **subsumption method** [3], in which a layered type of control is used, giving the behaviors different priority, **Distributed Architecture for Mobile Navigation** (DAMN) [18] in which behavioral selection is based on a voting system. Other methods include **activation networks** [9], **potential fields** [7], and **fuzzy command fusion** [15]. For a review of behavioral organization, see e.g. [1] and [14].

In this simulation package only a very simple behavioral selection mechanism is implemented. Since the package is oriented towards biologically inspired computation methods, similiar inspirations has guided the choice of method in the behavioral selection mechanism, which is based on the recently developed **utility function** method [19]. In this method, each behavior is assigned a (time-varying) utility value, which is determined based on sensory input and internal signals within the brain of the robot. At all times, the behavior associated with the highest utility value is selected for activation. Thus, the behavioral selection is completely determined by the shapes of the utility functions, which are optimized through artificial evolution.

3 The simulator

The simulation library consists of a dynamics engine, using the Featherstone algorithm for solving the equations of motion of tree-structured rigid body systems, a generic implementation of an evolutionary algorithm, and basic implementation of a behavioral selection mechanism.

Implemented in object-oriented Pascal, the library can be used in many different operating systems. The development of the library was done in the Delphi [2], an excellent environment for rapid application development, which is capable of producing fast executables.

This section starts with the description of the implemented dynamics engine and then proceeds by describing the parts related to behaviors and the evolutionary algorithm included in the simulation library. At the end of the section, the public interfaces of the different classes are listed.

3.1 The dynamics engine

3.1.1 Articulated body dynamics: The algorithm

The algorithm used here scales linearly with the number of degrees of freedom. Thus, each serial chain has a computational cost of $\mathcal{O}(N)$, where N is the number of degrees of freedom of the chain. These algorithms are often referred to as **articulated body** (AB) methods. Another group of algorithms commonly used are called **composite rigid body** (CRB) methods [20] and these methods have a computational cost of $\mathcal{O}(N^3)$. However, when N is small the CRB methods are more efficient than the AB method.

The AB dynamics algorithm is recursively formulated and can be divided into three basic parts:

1. Forward kinematics (outboard recursion)
   Computation of velocities and velocity dependent terms.

2. Backward dynamics (inboard recursion)
   Computation of articulated body inertia and bias forces (contact forces and applied torques).

3. Forward accelerations (outboard recursion)
   Computation of joint and link accelerations.

For a single serial chain with a fixed base, the three recursions in the AB dynamics algorithm are shown in Algorithm 1, Algorithm 2, and Algorithm 3. (See Appendix A for a description of the notation used.)

Algorithm 1 Forward Kinematics

Require: $\boldsymbol{\omega}_0 = 0, {}^i\mathbf{R}_{i-1}, {}^{i-1}\mathbf{p}_i, \dot{q}_i$
for $i = 1$ to NumberOfLinks **do**

$$\boldsymbol{\omega}_i = {}^i\mathbf{R}_{i-1}\boldsymbol{\omega}_{i-1} + \sigma_i\dot{q}_i\hat{\mathbf{z}} \tag{1}$$

$$\boldsymbol{\zeta}_i = \begin{bmatrix} \mathbf{0} \\ {}^i\mathbf{R}_{i-1}[\boldsymbol{\omega}_{i-1} \times (\boldsymbol{\omega}_{i-1} \times {}^{i-1}\mathbf{p}_i)] \end{bmatrix} + \begin{bmatrix} \sigma_i(\boldsymbol{\omega}_i \times \dot{q}_i\hat{\mathbf{z}}_i) \\ \bar{\sigma}_i(2\boldsymbol{\omega}_i \times \dot{q}_i\hat{\mathbf{z}}_i) \end{bmatrix} \tag{2}$$

$$\boldsymbol{\beta}_i = \begin{bmatrix} \boldsymbol{\omega}_i \times \bar{\mathbf{I}}_i\boldsymbol{\omega}_i \\ \boldsymbol{\omega}_i \times (\boldsymbol{\omega}_i \times \mathbf{h}_i) \end{bmatrix} \tag{3}$$

end for

Algorithm 2 Backward Dynamics

Require: $\tau_i, \mathbf{I}_N^\star = \mathbf{I}_N, \boldsymbol{\beta}_N^\star = \boldsymbol{\beta}_N - {}^{N+1}\mathbf{X}_N^{\mathrm{T}}\mathbf{f}_{N+1}$
for $i =$ NumberOfLinks to 1 **do**

$$\mathbf{n}_i = \mathbf{I}_i^\star\phi_i \tag{4}$$

$$m_i^\star = \boldsymbol{\phi}_i^{\mathrm{T}}\mathbf{I}_i^\star\boldsymbol{\phi}_i \tag{5}$$

$$\mathbf{N}_i = \mathbf{I}_i^\star - \mathbf{n}_i(m_i^\star)^{-1}\mathbf{n}_i^{\mathrm{T}} \tag{6}$$

$$\tau_i^\star = \tau_i + \boldsymbol{\phi}_i^{\mathrm{T}}\boldsymbol{\beta}_i^\star \tag{7}$$

$$\mathbf{I}_{i-1}^\star = \mathbf{I}_{i-1} + {}^i\mathbf{X}_{i-1}^{\mathrm{T}}\mathbf{N}_i{}^i\mathbf{X}_{i-1} \tag{8}$$

$$\boldsymbol{\beta}_{i-1}^\star = \boldsymbol{\beta}_{i-1} + {}^i\mathbf{X}_{i-1}^{\mathrm{T}}\left[\boldsymbol{\beta}_i^\star - \mathbf{N}_i\boldsymbol{\zeta}_i - \mathbf{n}_i(m_i^\star)^{-1}\tau_i^\star\right] \tag{9}$$

end for

Algorithm 3 Forward Accelerations

Require: $\mathbf{a}_0' = [\mathbf{0}^{\mathrm{T}} - {}^0\mathbf{a}_g^{\mathrm{T}}]^{\mathrm{T}}$
for $i = 1$ to NumberOfLinks **do**

$$\bar{\mathbf{a}}_i = {}^i\mathbf{X}_{i-1}\mathbf{a}'_{i-1} + \boldsymbol{\zeta}_i \tag{10}$$

$$\ddot{q}_i = (m_i^\star)^{-1}\tau_i^\star - \left[\mathbf{n}_i(m_i^\star)^{-1}\right]^{\mathrm{T}}\bar{\mathbf{a}}_i \tag{11}$$

$$\mathbf{a}_i' = \bar{\mathbf{a}}_i + \boldsymbol{\phi}_i\ddot{q}_i \tag{12}$$

end for

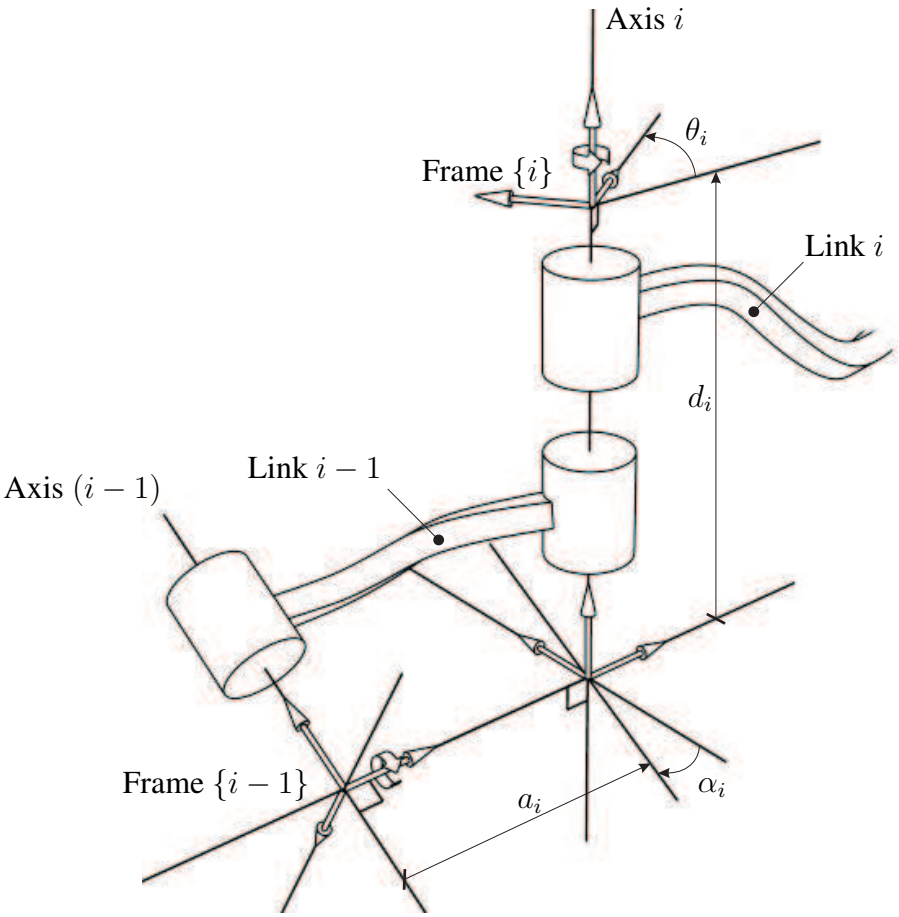


Figure 1: Definition of link frames and link parameters.

3.1.2 Link description

Using the recursive method presented in this report it is convenient to describe the relations between successive coordinate frames using only four parameters. This is possible since every joint in the tree-structured system only introduces one extra degree of freedom. The parameters needed are two offsets and two rotations (a, α, d, θ) and are commonly called the Denavit-Hartenberg (DH) parameters, named after their inventors. Depending on how successive frames are enumerated, they are sometimes called the *modified* Denavit-Hartenberg (MDH) parameters [4].

The assignment of link frames can be done in a number of ways and there is (usually) nothing unique about a certain configuration. For instance, when aligning the $\hat{Z}$-axis of a link's frame with the joint axis, the *direction* of $\hat{Z}$ may be chosen arbitrarily. When faced with multiple choices one (generally) choose the

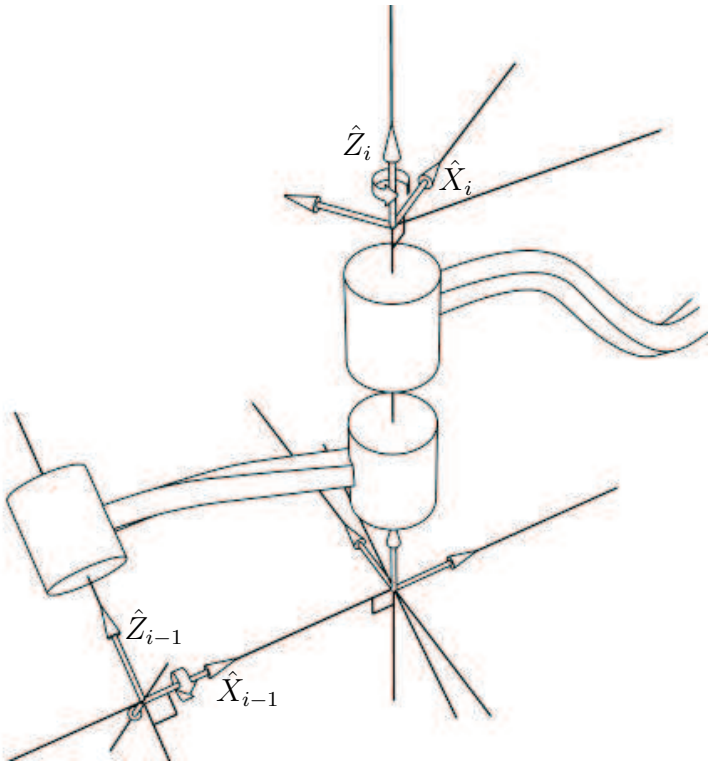


Figure 2: Link frames and their coordinate axes. The $\hat{Y}$-axis completes the right-handed coordinate system and is not indicated since it is not involved in the link-to-link transformations.

placement of the link frame such that it causes as many paramaters as possible to be zero, or simplifies the definition of the link's inertia matrix.

A procedure for assigning link frames is proposed in [4] and is reproduced here in a slightly modified form:

1. Identify the joint axes in the mechanism and imagine each axis as being infinitely long.
2. Enumerate the joints from 1 to N, where index 1 refers to the joint closest to the base of the mechanism and index N refers to the outermost joint (end effector). Index 0 is used for the base reference frame. These indices are later used to refer to the link frames.
3. Choose two adjacent joint axes (i and $i+1$).
4. Identify the common perpendicular between axis i and $i+1$. If the axes intersect, the point of intersection is identified.
5. Assign the origin of link frame $\{i\}$ to the point where the common perpendicular (between joint axes i and $i+1$) meets the i^{th} joint axis. If the joint axes intersect, place the origin of the i^{th} link frame at the point of intersection.
6. Assign the $\hat{Z}_i$-axis along joint axis i. The direction may be chosen arbitrarily.
7. Assign the $\hat{X}_i$-axis to lie along the common perpendicular, or, if the joint axes (i and $i+1$) intersect, assign $\hat{X}_i$ to be normal to the plane containing the axes. In the case of intersecting joint axes, the direction of $\hat{X}_i$ is arbitrary.
8. Assign the $\hat{Y}_i$-axis to complete the right-handed coordinate system.
9. Repeat step 3–8 until link frames $\{1\}$–$\{N-1\}$ have been assigned.
10. Assign frame $\{0\}$ (base frame) and frame $\{N\}$ (end effector) abitrarily, but preferably so as to cause as many parameters as possible to become zero (simplifies calculations).
11. Define the MDH parameters according to their definitions in (13).

Note that the procedure above does not handle cases where the mechanism contains branching points. However, this can be solved by applying the above procedure to each branch separately.

3.1.3 Modeling

In this section, the procedure of assigning link frames will be the described in further detail. Modeling a system is basically a matter of positioning link frames (local coordinate systems) and deriving parameters associated with them. In this section the *modified* Denavit-Hartenberg (MDH) convention will be used to describe these parameters (see [4]). At the end of this section an example of the modeling of a double pendulum is given.

Link frames are used to express parameters associated with a specific link, e.g. rigid body inertia, position of the link's center of mass, joint motion axis etc. By following certain guidelines (given below) when positioning the link frames, calculations are simplified and the computation cost will thus be reduced. When considering links with only a single degree of freedom, the origin of a link frame is usually placed such that one of the axes is aligned with the joint axis. This simplifies link-to-link transformations and computations involving the joint motion axis. It is also advantageous to place the link frame in a position such that the frame axes are coincident with the link's center of mass and aligned with the principal moments of inertia. This makes the spatial inertia matrix of the link diagonal, reducing the computation cost for the equations of motion. In this report the frames will be positioned with the $\hat{Z}$-axis of the link frame aligned with the joint axis. This approach is the most beneficial one from a computational point of view.

When using the MDH convention, the $\hat{Z}_i$-axis of the i^{th} link frame must be aligned with the i^{th} joint axis. In the case of a rotational joint, the $\hat{Z}$-axis is coincident with the joint's axis of rotation.

The link parameters used here are defined as follows (see Fig. 1):

$$\begin{array}{rcl} a_i & = & \text{the distance from } \hat{Z}_{i-1} \text{ to } \hat{Z}_i \text{ taken along } \hat{X}_{i-1} \\ \alpha_i & = & \text{the angle between } \hat{Z}_{i-1} \text{ and } \hat{Z}_i \text{ taken about } \hat{X}_{i-1} \\ d_i & = & \text{the distance from } \hat{X}_{i-1} \text{ to } \hat{X}_i \text{ taken along } \hat{Z}_i \\ \theta_i & = & \text{the angle between } \hat{X}_{i-1} \text{ and } \hat{X}_i \text{ taken about } \hat{Z}_i \end{array} \tag{13}$$

All angles are measured about the axes in a right-handed sense and the $\hat{Y}_i$-axes are defined so as to complete the right-handed coordinate systems.

□ EXAMPLE 3.1 Modeling of a double pendulum fixed in space (2 degrees of freedom).

(See Listing 2, p. 27 and Figure 3.)

First, the origin of the system (frame $\{0\}$) is positioned. In this case the system is placed

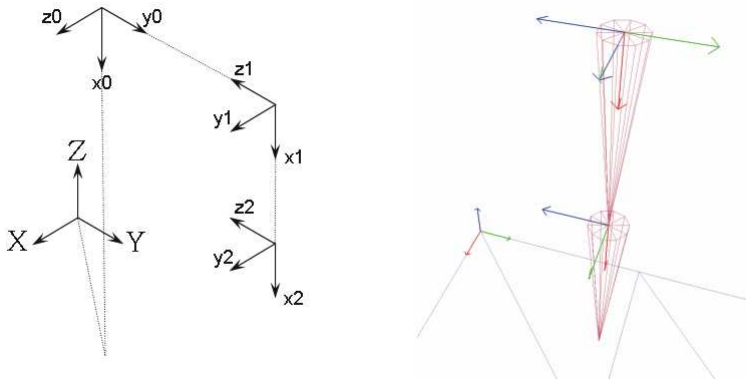


Figure 3: Model (left) and physical (right) configuration of the double pendulum. For clarity, the origins of frame $\{0\}$ and frame $\{1\}$ have been separated.

at the coordinate $[5.0, 5.0, 10.0]^{\mathrm{T}}$ and rotated $90°$ around the y-axis, using the quaternion $[0.0, 0.7071, 0.0, 0.7071]$. The rotation is done in order to prepare for the next step: aligning the $\hat{Z}$ axis with the first joint's axis of rotation (acc. to the specific model used).

Now, the first rotational joint is defined. In order to position frame $\{1\}$ acc. to Fig. 3, frame $\{0\}$ is rotated $90°$ around the x_0-axis. Thus, the MDH parameters for the first link are $[a_1, \alpha_1, d_1, \theta_1] = [0.0, 1.5708, 0.0, 0.0]$. Imagining that the length of each link is three units we can define the center of gravity for the first link to be located at $[1.5, 0.0, 0.0]^{\mathrm{T}}$ (relative to frame $\{1\}$).

Frame $\{2\}$ is just an offset away from frame $\{1\}$. Hence, the MDH parameters for the second (and last) link are defined as $[a_2, \alpha_2, d_2, \theta_2] = [3.0, 0.0, 0.0, 0.0]$.

Having a configuration as shown in Listing 2, the double pendulum would initially be in its stable (vertical) position. If the links should be positioned, let's say $45°$ from the global Z-axis, only θ_1 of the first link would have to be changed to be equal to $\pi/4 = 0.7854$. $\Diamond$

3.1.4 Integration method

The method used to integrate the equations of motion was a Runge-Kutta method with Cash-Karp parameters (see [16]). In this case a fourth order[1] method was used, with an error estimate (truncation error)[2] taken as the difference between a fourth and a fifth order approximation. This method takes a number of sampled slopes over an interval and then uses this information to advance the solution. In contrast with e.g. the backward Euler method, no prior behavior of the solution is used in its propagation.

[1] A method is conventionally called n^{th} order if its error term is $O\left(h^{n+1}\right)$.
[2] Error induced by the method itself.

The formula for advancing the solution from time t to $t+h$, where h is the step size, is

$$\mathbf{y}_{k+1} = \mathbf{y} + h\left(\tfrac{37}{378}\mathbf{k}_1 + \tfrac{250}{621}\mathbf{k}_3 + \tfrac{125}{594}\mathbf{k}_4 + \tfrac{512}{1771}\mathbf{k}_6\right) \ , \tag{14}$$

where the sampled slopes with Cash-Karp parameters are

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}\left(t, \mathbf{y}\right) \\
\mathbf{k}_2 &= \mathbf{f}\left(t + \tfrac{1}{5}h, \mathbf{y} + \left(\tfrac{1}{5}h\mathbf{k}_1\right)\right) \\
\mathbf{k}_3 &= \mathbf{f}\left(t + \tfrac{3}{10}h, \mathbf{y} + h\left(\tfrac{3}{40}\mathbf{k}_1 + \tfrac{9}{40}\mathbf{k}_2\right)\right) \\
\mathbf{k}_4 &= \mathbf{f}\left(t + \tfrac{3}{5}h, \mathbf{y} + h\left(\tfrac{3}{10}\mathbf{k}_1 - \tfrac{9}{10}\mathbf{k}_2 + \tfrac{6}{5}\mathbf{k}_3\right)\right) \\
\mathbf{k}_5 &= \mathbf{f}\left(t + h, \mathbf{y} + h\left(-\tfrac{11}{54}\mathbf{k}_1 + \tfrac{5}{2}\mathbf{k}_2 - \tfrac{70}{27}\mathbf{k}_3 + \tfrac{35}{27}\mathbf{k}_4\right)\right) \\
\mathbf{k}_6 &= \mathbf{f}\left(t + \tfrac{7}{8}h, \mathbf{y} + h\left(\tfrac{1631}{55216}\mathbf{k}_1 + \tfrac{175}{512}\mathbf{k}_2 + \tfrac{575}{13824}\mathbf{k}_3 + \tfrac{44275}{110592}\mathbf{k}_4 + \tfrac{253}{4096}\mathbf{k}_5\right)\right) \ .
\end{aligned} \tag{15}$$

The estimate of the local truncation error is calculated as

$$\mathbf{err} = h\left(D_1\mathbf{k}_1 + D_3\mathbf{k}_3 + D_4\mathbf{k}_4 + D_5\mathbf{k}_5 + D_6\mathbf{k}_6\right) \tag{16}$$

with the coefficients

$$\begin{cases}
D_1 &= \tfrac{37}{378} - \tfrac{2825}{27648} \\
D_3 &= \tfrac{250}{621} - \tfrac{18575}{48384} \\
D_4 &= \tfrac{125}{594} - \tfrac{13525}{55296} \\
D_5 &= -\tfrac{277}{14336} \\
D_6 &= \tfrac{512}{1771} - \tfrac{1}{4} \ .
\end{cases} \tag{17}$$

The error estimate can be used to monitor the accuracy of the solution. In adaptive routines, this estimate is used to control the step size. If an error estimate is above a certain tolerance level, the step size is decreased and the solution is re-calculated, using the new step size. It is also possible to increase the step size if the error estimate is below a certain tolerance level.

The simulation library currently only supports a constant step size, which is acceptable as long as the step size is taken sufficiently small. A typical step size ranges from 0.001 to 0.01.

3.1.5 Contact model

In the current implementation of the simulation library, ground contact forces are modeled by a spring/damper system. The model contains two pairs of spring/-damper models, one in the normal direction of the contact point and one in the

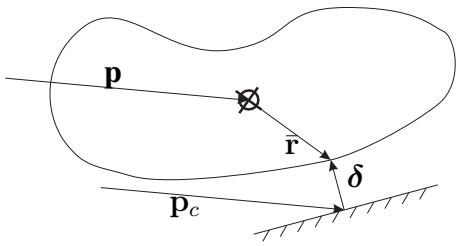


Figure 4: Checking for contacts.

tangent plane. This structure requires the assignment of four constants that define the stiffness and the damping coefficients. Assuming that a contact point ($\bar{\mathbf{r}}$) is defined with respect to a body's coordinate system, a collision is determined by calculating the vector $\boldsymbol{\delta}$ (see Fig. 4) as

$$\boldsymbol{\delta} = (\mathbf{p} + \mathbf{R}^{\mathrm{T}}\bar{\mathbf{r}}) - \mathbf{p}_c \quad , \tag{18}$$

where $\mathbf{R}$ is the body's orientation (rotation) matrix. If $\boldsymbol{\delta}$ is negative then the two bodies are in contact and the forces generated by the springs and dampers need to be calculated

In order to compute the damping force, the translational velocity of the contact point is needed. Since this contact is fixed to the current body, all that is needed is to calculate the local velocity and then apply the current link's rotational matrix, transforming it from the local coordinate system to the inertial coordinate system. The following equation yields the translational velocity of the contact point (w.r.t. the inertial coordinate system):

$$\mathbf{v}_c = \mathbf{R}^{\mathrm{T}} \underbrace{(\bar{\mathbf{v}} + \boldsymbol{\omega} \times \bar{\mathbf{r}})}_{\bar{\mathbf{v}}_c} \quad , \tag{19}$$

where $\bar{\mathbf{r}}$ is the position of the contact point with respect to the link's local coordinate system (see Fig. 5) and $\mathbf{R}$ is the link's orientation matrix. Note that the rotation matrix $\mathbf{R}$ produces a transformation from the inertial coordinate system to the local coordinate system.

Calculating the force in the normal direction is a simple matter of taking the negative normal components of the spring displacement vector ($\boldsymbol{\delta}$) and the velocity of the contact point ($\mathbf{v}_c$) and scaling them with their respective coefficients as

$$\mathbf{F}_n = -(k_n(\boldsymbol{\delta}^{\mathrm{T}}\mathbf{n}) + d_n(\mathbf{v}^{\mathrm{T}}\mathbf{n}))\mathbf{n} \quad , \tag{20}$$

where k_n and d_n are the coefficients in the normal direction for the spring and damper, respectively. In order for the normal force to be valid it must have a

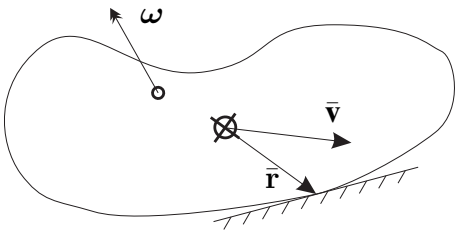


Figure 5: Calculating the velocity of the body at the contact point.

magnitude greater than zero ($\|\mathbf{F}_n\| > 0$), otherwise it would correspond to the unphysical case of objects attracting each other.

The force planar to the collision surface is calculated by multiplying the planar velocity vector and the planar displacement vector with their respective coefficients: the planar damping coefficient (d_p) and the planar spring coefficient (k_p), as

$$\mathbf{F}_p = -(k_p(\boldsymbol{\delta} - \mathbf{n}(\boldsymbol{\delta}^{\mathrm{T}}\mathbf{n})) + d_p(\mathbf{v} - \mathbf{n}(\mathbf{v}^{\mathrm{T}}\mathbf{n}))). \tag{21}$$

Once the planar and normal forces have been calculated, the contact point is checked for sliding or sticking, i.e. whether to use the static friction coefficient (μ_s) or the kinetic friction coefficient (μ_k). This is determined by calculating the ratio between the magnitudes of the planar and normal force vectors as

$$\begin{cases} \text{if} \quad \|\mathbf{F}_p\|/\|\mathbf{F}_n\| < \mu_k \quad \text{then slide,} \\ \text{if} \quad \|\mathbf{F}_p\|/\|\mathbf{F}_n\| > \mu_s \quad \text{then stick.} \end{cases} \tag{22}$$

If sliding is indicated by Eq. (22) then the planar force is re-calculated in order to satisfy $\mu_k = \|\mathbf{F}_f\|/\|\mathbf{N}\|$, where $\mathbf{F}_f$ is the friction force and $\mathbf{N}$ is the normal force.

$$\hat{\mathbf{F}}_p = \begin{cases} \frac{\|\mathbf{F}_n\|}{\|\mathbf{F}_p\|}\mu_k\mathbf{F}_p & \text{if sliding} \\ \mathbf{F}_p & \text{if not sliding} \end{cases} \tag{23}$$

Finally, the normal and planar force components are added and transformed from the inertial coordinate system to the colliding body's local coordinate system as

$$\bar{\mathbf{F}} = \mathbf{R}(\mathbf{F}_n + \hat{\mathbf{F}}_p) \quad . \tag{24}$$

The resultant spatial force, affecting the colliding body, is then calculated as

$$\mathbf{f}_c = \begin{bmatrix} \bar{\mathbf{r}} \times \bar{\mathbf{F}} \\ \bar{\mathbf{F}} \end{bmatrix} \quad , \tag{25}$$

where the upper part of $\mathbf{f}_c$ is the torque vector, with respect to the body's local coordinate system. If there are multiple contact points (at the same time), the resultant spatial force is the sum of all contact forces.

3.1.6 Software interfaces

In this section, the interfaces of the most important classes included in the library, i.e. the parts that are accessible to the user (see Appendix C), are listed. Most of the methods and variables shown in these listings are named in a self-explanatory manner. However, in cases where they are not, a description of the method (or variable) is given in connection with the listing. All listings are given in object-oriented Pascal.

TABSimulation This class provides the main interface to the dynamics engine. It supplies methods for visualizing the system, querying the contact sensors, retrieving link information, setting joint inputs etc.

Filename: `ABSimulation.pas`
Inheritance: `TABSimulation = class(TObject)`
Interface:

```
1  TABSimulation = class(TObject)
2  public
3    {Public Declarations}
4    constructor Create(const AFile: string); overload;
5    constructor Create(const AStream: TStream); overload;
6    destructor Destroy; override;
7
8    procedure GetCOM(var pos: TCartesianVector);
9    procedure GetContactForces(MemberName: string; ContactIndex:
         Cardinal;
10     var Fx, Fy, Fz: ABFloat);
11   function GetContactSensorValueZ(MemberName: string;
12     ContactIndex: Cardinal): ABFloat;
13   procedure GetLinkAccelerationAtPos(LinkName: string;
14     const RelativePos: TCartesianVector; var AccXYZ:
          TCartesianVector);
15   procedure GetLinkCOMState(LinkName: string; var Pos, Vel:
        TCartesianVector);
16   procedure GetLinkState(LinkName: string; var State:
        TDynVector);
17   function  GetRevLinkName(Index: Cardinal): string;
18   procedure Reset;
19   procedure Simulate;
20   procedure SimulateWithGraphics;
21   procedure SetRevJointInput(LinkName: string; Input: ABFloat)
        ;
22   procedure SetTransJointInput(LinkName: string; Input:
        ABFloat);
23   procedure SetZeroInput;
24   procedure Terminate;
```

```
25      procedure UpdateContacts;
26
27      procedure DisplayAllMembers;
28      procedure DisplayContactLinkNames;
29      procedure DisplayPrismaticLinkNames;
30      procedure DisplayRevoluteLinkNames;
31
32      property ContactLinkNames: TStringList read Fcp_link_names;
33      property SimulationSteps: Cardinal read Fsteps write Fsteps;
34      property StepSize: ABFloat read Fdt write Fdt;
35      property System: TSystem read Fsystem;
36      property MaxSteps: Cardinal read Fmax_steps write Fmax_steps
            ;
37      property MemberNames: TStringList read Fsys_member_names;
38      property NumContacts: Cardinal read Fnum_contact_points;
39      property NumDOFs: Cardinal read Fsys_dof;
40      property NumRevoluteJoints: Cardinal read Fnum_rev_joints;
41      property NumTranslationalJoints: Cardinal read
            Fnum_trans_joints;
42      property RevoluteLinkNames: TStringList read
            Frev_joint_names;
43      property PrismaticLinkNames: TStringList read
            Ftrans_joint_names;
44
45      property OnDoStep: TNotifyEvent read FOnDoStep write
            FOnDoStep;
46    end;
```

TDynObject This is an abstract base class (see Appendix C) common to all objects in the simulation environment. Currently, it only stores a name for each object.

Filename: `AB_BaseObject.pas`
Inheritance: `TDynObject = class(TObject)`
Interface:

```
1   TDynObject = class(TObject)
2   public
3     constructor Create; virtual;
4     destructor Destroy; virtual;
5     procedure SetName(const AName: string);
6     function GetName: string;
7     property UserData[Index: integer]: TUserData read
          GetUserData
8       write SetUserData;
9   end;
```

TEnvironment This class encapsulates everything needed for the interaction with the environment, such as the gravity vector and height data describing the terrain.

Filename: `AB_Environment.pas`
Inheritance: `TEnvironment = class(TDynObject)`
Interface:

```
1  TEnvironment = class(TSystem)
2  public
3    constructor Create; override;
4    destructor Destroy; override;
5
6    procedure SetEnvironment(env: TEnvironment);
7    function GetEnvironment: TEnvironment;
8    procedure LoadTerrainData(FileName: string);
9    function GetTerrainData(var xdim, ydim: integer;
10     var spacing: ABFloat): TDynMatrix2D;
11   procedure SetGravity(const gravity: TCartesianVector);
12   procedure GetGravity(var gravity: TCartesianVector);
13 end;
```

TForce This is an abstract base class for all instantiated force objects (external forces). Derived classes are responsible for the implementation of all the class methods in `TForce`. The method `ComputeForce` returns the spatial force vector defined as

$$\mathbf{f} = \begin{bmatrix} \mathbf{n} \\ \boldsymbol{f} \end{bmatrix},$$

where $\mathbf{n}$ is the 3×1 torque vector and $\boldsymbol{f}$ is the 3×1 force vector.

Filename: `AB_Force.pas`
Inheritance: `TForce = class(TDynObject)`
Interface:

```
1  TForce = class(TDynObject)
2  public
3    constructor Create; override;
4    destructor Destroy; override;
5
6    procedure ComputeForce(const val: TABForwardKinematics;
7      var force: TSpatialVector); virtual; abstract;
8    function GetNumContactPoints: Cardinal; virtual; abstract;
9    procedure GetContactForces(var force: TCartesianArray;
10     const offset: integer); virtual; abstract;
11 end;
```

TMDHLink Class of links with one degree of freedom (revolute and prismatic), specified by the modified Denavit-Hartenberg (MDH) parameters (see Fig. 1, p. 10). This is the base class for the two kinds of single DOF link classes: `TRevoluteLink` and `TPrismaticLink`. These two descendant classes (see Appendix C) are responsible for implementing any abstract methods in this class.

Filename: `AB_MDHLink.pas`
Inheritance: `TMDHLink = class(TRigidBody)`
Interface:

```
1  TMDHLink = class(TRigidBody)
2  public
3    constructor Create; override;
4    destructor Destroy; override;
5
6    procedure SetMDHParameters(a, alpha, d, theta: ABFloat);
7    procedure GetMDHParameters(var a, alpha, d, theta: ABFloat);
8    procedure SetJointLimits(min, max, spring, damper: ABFloat);
9    procedure GetJointLimits(var min, max, spring, damper:
        ABFloat);
10   function GetNumDOFs: integer; override;
11   procedure SetState(const q, qd: TDynVector;
12     q_offset, qd_offset: integer); override;
13   procedure GetState(var q, qd: TDynVector;
14     q_offset, qd_offset: integer); override;
15   procedure SetJointInput(const JointInput: TDynVector;
16     const offset: integer); override;
17
18   procedure ABForwardKinematics(var q, qd: TDynVector;
19     q_offset, qd_offset: integer;
20     const inboard_link: TABForwardKinematics;
21     var current_link: TABForwardKinematics); override;
22   procedure ABBackwardDynamics(var current_link:
        TABForwardKinematics;
23     const f_curr: TSpatialVector;
24     const N_curr: TSpatialTensor;
25     var f_inboard: TSpatialVector;
26     var N_inboard: TSpatialTensor); override;
27   procedure ABForwardAccelerations(const acc_inboard:
        TSpatialVector;
28     var acc_current: TSpatialVector;
29     var qd, qdd: TDynVector;
30     qd_offset, qdd_offset: integer); overload; override;
31 end;
```

TPrismaticLink This is one of the two concrete descendants of `TTransform`, i.e. implementing all parts of the inherited class (see Appendix C). It implements the dynamics specific to a one degree of freedom translational joint. The main reason for this class is to implement an optimized version of the spatial congruence transformation of the AB inertia matrix.

Filename: `AB_PrismaticLink.pas`
Inheritance: `TPrismaticLink = class(TMDHLink)`
Interface:

```
1  TPrismaticLink = class(TMDHLink)
2  public
3    constructor Create; override;
4    destructor Destroy; override;
5
6    procedure TransformationToInboard(const N: TSpatialTensor;
7      var I: TSpatialTensor); override;
8  end;
```

TRevoluteLink Together with `TPrismaticLink` this is the second of the two concrete descendants of `TTransform`. It implements the dynamics specific to a one degree of freedom translational joint. The main reason for this class is to implement an optimized version of the spatial congruence transformation of the AB inertia matrix.

Filename: `AB_RevoluteLink.pas`
Inheritance: `TRevoluteLink = class(TMDHLink)`
Interface:

```
1  TRevoluteLink = class(TMDHLink)
2  public
3    constructor Create; override;
4    destructor Destroy; override;
5
6    procedure TransformationToInboard(const N: TSpatialTensor;
7      var I: TSpatialTensor); override;
8  end;
```

TRigidBody This class contain all the dynamic properties for rigid bodies that are needed for the AB algorithm. Together with the class `TTransform`, which implement the different transformations needed, it is equivalent to a link object.

Filename: `AB_RigidBody.pas`

Inheritance: `TRigidBody = class(TTransform)`
Interface:

```
TRigidBody = class(TTransform)
public
  constructor Create; override;
  destructor Destroy; override;

  function SetInertiaParameters(mass: ABFloat;
    const Inertia: TCartesianTensor;
    const COM_pos: TCartesianVector): boolean;
  procedure GetInertiaParameters(var mass: ABFloat;
    var Inertia: TCartesianTensor;
    var COM_pos: TCartesianVector);

  function GetNumForces: Cardinal;
  procedure GetCenterOfMassPos(link_values:
     TABForwardKinematics;
    var p_COM: TCartesianVector);
  procedure GetCenterOfMassVel(link_values:
     TABForwardKinematics;
    var v_COM: TCartesianVector);
  function GetPotentialEnergy(link_values:
     TABForwardKinematics;
    const gravity: TCartesianVector): ABFloat; override;
  function GetKineticEnergy(link_values: TABForwardKinematics)
     : ABFloat;
    override;
  function GetNumContactPoints: Cardinal; override;
  procedure GetContactForces(var forces: TCartesianArray;
    const offset: integer); override;

  property Mass: ABFloat read Fmass;
end;
```

TSystem This is an abstract base class and contains references to each serial chain (`TTreeStructure`) present in the system. The class method `Dynamics` performs the recursions needed for each serial chain and returns the derivatives of the system's state variables. The derivatives are then passed to a suitable integrator and then the state variables of the system are updated accordingly.

Filename: `AB_System.pas`
Inheritance: `TSystem = class(TDynObject)`
Interface:

```
TSystem = class(TDynObject)
public
```

```
3     constructor Create; override;
4     destructor Destroy; override;
5
6     function GetNumDOFs: Cardinal; virtual; abstract;
7     procedure SetState(const q, qd: TDynVector;
8       const pos_offset, vel_offset: integer); virtual; abstract;
9     procedure GetState(var q, qd: TDynVector;
10      q_offset, qd_offset: integer); virtual; abstract;
11
12    procedure InitVariables(var qy, qdy: TDynVector);
13
14    function GetPotentialEnergy: ABFloat; virtual; abstract;
15    function GetKineticEnergy: ABFloat; virtual; abstract;
16    procedure GetCOM(var pos: TCartesianVector); virtual;
          abstract;
17
18    procedure Dynamics(var qy, qdy: TDynVector); virtual;
          abstract;
19  end;
```

TTransform Base class for all of objects that apply any kind of transformation in the serial chain. Together with the class `TRigidBody` it defines a complete link object, with transformations and dynamic properties.

This class implements all the necessary (common) transformations such as e.g. rotational transformations and spatial transformations (both inwards and outwards).

Filename: `AB_Transform.pas`
Inheritance: `TTransform = class(TDynObject)`
Interface:

```
1   TTransform = class(TDynObject)
2   public
3     constructor Create; override;
4     destructor Destroy; override;
5
6     function GetNumDOFs: integer; virtual; abstract;
7     procedure SetState(const q, qd: TDynVector;
8       q_offset, qd_offset: integer); virtual; abstract;
9     procedure GetState(var q, qd: TDynVector;
10      q_offset, qd_offset: integer); virtual; abstract;
11    procedure SetJointInput(const JointInput: TDynVector;
12      const offset: integer); virtual; abstract;
13
14    procedure ABForwardKinematics(var q, qd: TDynVector;
15      q_offset, qd_offset: integer;
```

```
16      const inboard_link: TABForwardKinematics;
17      var curr_link: TABForwardKinematics); virtual; abstract;
18    procedure ABBackwardDynamics(var current_link:
          TABForwardKinematics;
19      const f_curr: TSpatialVector;
20      const N_curr: TSpatialTensor;
21      var f_inboard: TSpatialVector;
22      var N_inboard: TSpatialTensor); virtual; abstract;
23    procedure ABForwardAccelerations(const a_inboard:
          TSpatialVector;
24      var a_current: TSpatialVector;
25      var qd, qdd: TDynVector;
26      qd_offset, qdd_offset: integer); overload; virtual;
            abstract;
27    function GetPotentialEnergy(const link_values:
          TABForwardKinematics;
28      const gravity: TCartesianVector): ABFloat; virtual;
            abstract;
29    function GetKineticEnergy(const link_val:
          TABForwardKinematics): ABFloat;
30      virtual; abstract;
31
32    function GetNumContactPoints: Cardinal; virtual; abstract;
33    procedure GetContactForces(var forces: TCartesianArray;
34      const offset: integer); virtual; abstract;
35  end;
```

3.1.7 Usage

In order to load a specific system, a configuration file must exist which follows a certain structure. Continuing Example 3.1, the configuration file for the double pendulum is shown in Listing 2.

Pseudo-code for simulating a system is as follows:

```
(1) read the system configuration file and create the system
(2) read the environment data and create the environment
(3) For each time step
      * simulate the system (integrate)
      * update sensors
      * update motor signals
      * set system inputs
```

Example code for performing the sequence above is shown in Listing 1, where the first two lines of code (line 12 and 13) creates the system (tree-structured) and the environment. The environement is created for purposes of providing the graphics

and for providing the contact model with ground elevation data. Some variables in the listing have a name including the letters CM, or COM. Those letters refer to the *center-of-mass* of one of the links in the system.

Listing 1: Example code for simulation of a system.

```
procedure TAgent.Evalute;
var
 Time: ABFloat;
 TimeStep: ABFloat;
 Nsteps: integer;
 j: integer;
 tree: TTreeStructure;
 environment: TEnvironment;
 CMPosition, CMVelocity: TCartesianVector;
 InitialCMPosition,InitialCMVelocity: TCartesianVector;
begin
 tree := LoadFile(SYSTEM_FILE);
 environment := LoadEnvironment(ENVIRONMENT_FILE);
 TimeStep := SimulationParameters.TimeStep;
 tree := TTreeStructure(System);
 tree.GetLinkCOM(2, InitialCMPosition, InitialCMVelocity);
 Nsteps := SimulationParameters.Nsteps;
 Time := 0.0;

 for j := 1 to Nsteps do
  begin
   tree.GetLinkCOM(2, CMPosition, CMVelocity);
   with (SensorReadings) do
   begin
     SetSensor(1,fContact_force[1][1][3]);
     SetSensor(2,fContact_force[1][2][3]);
     SetSensor(3,fContact_force[1][3][3]);
     SetSensor(4,fContact_force[1][4][3]);
     SetSensor(5,CMVelocity[1]);
     SetSensor(6,CMVelocity[2]);
     SetSensor(7,CMVelocity[3]);
   end;
   SensoryPreProcessingSystem.GenerateNewReadings(
     SensorReadings);
   RNN.Step(SensoryPreprocessingSystem.Readings,
     TimeStep);
   UpdateMotorSignals(Time);
   joint_torque[1] := MotorSignals[1];
   Joint_torque[2] := MotorSignals[2];
   SetJointInput(joint_torque);
   integrator.Simulate(TimeStep);
   UpdateContactForces;
   Time := Time + TimeStep;
```

```
  end;

  tree.Free;
  environment.Free;
end;
```

Listing 2: Configuration file for a double pendulum (see Example 3.1, p. 13).

```
# Configuration file for a double pendulum.

object Pendulum: TTreeStructure
  Position = {5.0, 5.0, 10.0}
  Orientation_Quat = {0.0, 0.7071, 0.0, 0.7071} #{x,y,z,w}

  object Link1: TRevoluteLink
    Graphics_Model = Cylinder_X
    Mass = 0.5
    Inertia = {
      0.1, 0.0, 0.0
      0.0, 1.7, 0.0
      0.0, 0.0, 1.7}
    Center_of_Gravity = {1.5, 0.0, 0.0}
    MDH_Parameters = {0.0, 1.5708, 0.0, 0.0} #{a,alpha,d,theta}
    Initial_Joint_Velocity = 0.0
    Joint_Friction = 0.35
  end

  object Link2: TRevoluteLink
    Graphics_Model = Cylinder
      Cylinder_Start_Point = {0.0, 0.0, 0.0}
      Cylinder_End_Point = {3.0, 0.0, 0.0}
      Cylinder_Start_Radius = 0.3
      Cylinder_End_Radius = 0.2
      Cylinder_Slices = 20

    Mass = 0.5
    Inertia = {
      0.1, 0.0, 0.0
      0.0, 1.7, 0.0
      0.0, 0.0, 1.7}
    Center_of_Gravity = {1.5, 0.0, 0.0}
    MDH_Parameters = {3.0, 0.0, 0.0, 0.0}
    Initial_Joint_Velocity = 0.0
    Joint_Friction = 0.35
  end

end #TreeStructure
```

3.2 Behaviors and evolutionary algorithm

In this library, behaviors are generated and, to a limited extent, organized using evolutionary algorithms (EAs).

In general, an agent (e.g. a simulated robot) consists of a body, a brain, and a genome which, when decoded, generates the body and the brain. However, the agents presently included in this package (see the description of the `TAgent` class below), are designed solely for generating simple motor behaviors, and thus have a simplified structure.

EAs is the common term used for algorithms inspired by natural evolution. Examples of such algorithms are: genetic algorithms (GAs) [6], genetic progamming (GP) [8], and evolution strategies (ES) [17]. EAs operate on a population of candidate solutions and apply genetic operators (selection, mutation, crossover) to evolve the solutions and to improve their performance. The encoding of a candidate solution can be made in many different ways, such as, strings of digits, arrays, lists, and trees. It is also possible to implement an EA in such a way that the genetic operators act directly on the target structure, i.e. without using the intermediate steps of encoding and decoding. Regardless of the representation used, the genetic operators must of course be defined such that they are able to operate on that representation, i.e. such that they always generate valid structures.

EAs are suitable methods when searching through large spaces with many local optima [12]. It is also possible to mix both continuous and discrete optimization as long as it is possible to formulate a fitness (objective) function. Another benefit is the possibility to use EAs in cases where it is difficult to derive an analytical model of the system, or when such a model simply does not exist [21].

A description of the classes defining the behavioral architectures, the agents, and the EA, will now be given, followed by usage examples.

3.2.1 Behavioral architecture

In this simulation library, only two types of behavioral architectures are currently included, namely a PD controller and a recurrent neural network (RNN).

TPDController The PD controller can take any number of inputs and produce an output based on reference values, proportional gains, and damping constants. Each output from the PD controller is generated as follows

$$y = K_p(x_{ref} - x) + K_d(\dot{x}_{ref} - \dot{x}), \tag{26}$$

where K_p, K_d are the proportional and derivative constants, respectively, and x and $\dot{x}$ are the inputs (cf. state variables) to the controller.

Filename: `PDController.pas`
Inheritance: `TPDController = class(TObject)`
Interface:

```
1  TPDController = class(TObject)
2  public
3    {Public declarations}
4    constructor Create(NumberOfInputs, NumberOfOutputs: integer)
       ;
5    destructor Destroy; override;
6
7    procedure GenerateOutput(X, dX: TRealArray);
8    procedure SetParameters(Kp, Kd: TRealArray);
9    procedure SetKp(i: integer; Value: real);
10   procedure SetKd(i: integer; Value: real);
11   procedure SetReferenceValues(XRefValues, dXRefValues:
       TRealArray);
12   property Output[i: integer]: real read GetVal; default;
13   property NumberOfInputs: integer read fNumInputs;
14   property NumberOfOutputs: integer read fNumOutputs;
15   property NumberOfVariables: integer read fNumVariables;
16 end;
```

TRNN The recurrent neural network consists of a set of neurons and connection weights between them. In addition, each network has a number of inputs, which are connected to the neurons (some which might be output neurons) by the entries in $\mathbf{w}^{\text{IN}}$ (see class listing below). In the implementation used here, for a network with N neurons, N_{out} outputs, and N_{in} inputs, the size of the matrix W is $N \times N$, and the first N_{out} neurons are taken as the output neurons. Thus, the number of internal neurons equals $N - N_{\text{out}}$. The size of the $\mathbf{w}^{\text{IN}}$ matrix equals $N \times N_{\text{in}}$, making it possible to connect each input to every neuron in the network (including the output neurons).

For each neuron i, a time constant τ_i is defined, and the dynamical equations for the neurons are

$$\tau_i \dot{x}_i + x_i = g(b_i + \sum_{j=1}^{N} w_{ij} x_j + \sum_{j=1}^{N_{in}} w_{ij}^{\text{IN}} I_j), \tag{27}$$

where $g(z) = \tanh(cz)$ is the activation function (c is a constant), x_i is the activation level of neuron i, and I_j denotes input signal j. The b_i are bias terms which determine the output of the neuron in the absence of external inputs.

Filename: `RNN.pas`

Inheritance: `TRNN = class(TObject)`
Interface:

```
TRNN = class(TObject)
public
 constructor Create(NumberOfInputs, NumberOfOutputs,
    NumberOfNeurons: integer; W, Win: TMatrix; Tau, Bias:
    TVector); overload;
 procedure SetSigmoidParameter(c: real);
 procedure Step(Input: TVector; Dt: real);
 destructor Destroy; override;
 property M[i: integer]: real read GetVal; default;
 property NumberOfNeurons: integer read fNumberOfNeurons;
 property NumberOfInputs: integer read fNumberOfInputs;
 property NumberOfOutputs: integer read fNumberOfOutputs;
 property W: TMatrix read fW;
 property Win: TMatrix read fWin;
 property Tau: TVector read fTau;
 property Bias: TVector read fBias;
end;
```

3.2.2 Evolutionary algorithm

The evolutionary algorithm included in the library operates on a population of genomes (of the type `TGenome`). As mentioned above, many different representations can be chosen for the genomes, and the class `TGenome` must therefore, in general, be defined by the user. The specific case of genomes suitable for the evolution of RNNs has been implemented in the package. It is also required, from the point of view of the EA, that the user implements both the mutation and the crossover operator in the interface of the `TGenome` class.

Filename: `EA.pas`
Inheritance: `TEA = class(TObject)`
Interface:

```
TEA = class(TObject)
public
 constructor Create; overload;
 function FindBestIndividual: integer;
 procedure SetTournamentSelectionParameter(PTournament: real);
 procedure SetTournamentSize(Size: integer);
 procedure MakeNewGeneration;
 function TournamentSelect: integer; overload;
 function TournamentSelect(Population: TPopulation): integer;
    overload;
 destructor Destroy; override;
```

```
11   property Population: TPopulation read fPopulation write
         fPopulation;
12  end;
```

TGenome The listing below shows the procedures that the user must define. The `TGenomeParameters` class holds the various parameters associated with the genome, such as e.g. mutation rates.

Filename: `TGenome.pas`
Inheritance: `TGenome = class(TObject)`
Interface:

```
1  TGenome = class(TObject)
2  public
3    constructor CreateRandom(GenomeParameters: TGenomeParameters);
4    constructor CreateAndSet(Genome: TGenome);
5    procedure SetGenome(Genome: TGenome);
6    procedure Mutate;
7    class procedure Crossover(const X, Y: TGenome;
8      var Child1, Child2: TGenome);
9    destructor Destroy; override;
10   property M[i: integer]: real read GetVal write SetVal; default
         ;
11   property NumberOfGenes: integer read fNumberOfGenes;
12   property GenomeParameters: TGenomeParameters read
         fGenomeParameters
13     write fGenomeParameters;
14 end;
```

3.2.3 Agent

Objects of the `TAgent` class contain a genome, a brain (of type TRNN or TPD-Controller), a set of motor signals, and a set of sensor signals. The robotic brain is generated during the decoding of the genome. An example of a public interface of an agent is shown in the listing below.

Filename: `TAgent.pas`
Inheritance: `TAgent = class(TObject)`
Interface:

```
1  TAgent = class(TObject)
2  public
3   constructor Create; overload;
4   procedure SetGenome(Genome: TGenome);
```

```
5    procedure DecodeGenome;
6    procedure UpdateMotorSignals(Inputs: TVector; Time: real);
7    destructor Destroy; override;
8   end;
```

3.2.4 Usage

Below follows a short code example, showing the main loop of the evolutionary algorithm. The first three lines of code concerns the initialization of the evolutionary algorithm. The argument `EAParameters` is a variable containing all the parameters needed by the EA, such as tournament size and tournament selection probability, and is typically set from a graphical user interface (GUI).

Basic code, needed for evaluating all the individuals in the population is shown in Listing 3. First, an instance of the EA is created and its parameter are set by the variable `EAParameters`. Then, a random population is created, where each individual gets the properties as defined in the variable `GenomeParameters`. Finally the main loop starts, evaluating each individual (agent), creating a new population, and continues for as long as a termination criteria is not met.

Listing 3: Main loop in the evolutionary algorithm

```
1  EA := TEA.Create;
2  EA.SetParameters(EAParameters);
3  EA.Population := TPopulation.CreateRandom(PopulationSize,
       GenomeParameters);
4
5  repeat
6    for i := 1 to EA.Population.PopulationSize do
7    begin
8      Agent := TAgent.Create;
9      Agent.SetGenome(EA.Population[i]);
10     Agent.DecodeGenome;
11     Agent.Evaluate;
12     EA.Population.SetFitness(i, Agent.Fitness);
13     Agent.Free;
14   end;
15   EA.MakeNewGeneration;
16 until (Terminate);
17
18 EA.Free;
```

Appendices

Appendix A Notation

$\tilde{\mathbf{v}}$ Skew-symmetric 3×3 matrix constructed from the vector $\mathbf{v}$ such that $\tilde{\mathbf{v}}\mathbf{w} = \mathbf{v} \times \mathbf{w}$

$$\tilde{\mathbf{v}} = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

${}^{i-1}\mathbf{p}_i$ Relative position vector from the origin of frame $\{i-1\}$ to the origin of frame $\{i\}$.

${}^{i-1}\mathbf{v}_i$ 6×1 spatial velocity vector in frame $\{i\}$ expressed in frame $\{i-1\}$. A spatial velocity vector is composed by the 3×1 angular velocity vector $\boldsymbol{\omega}$ and the 3×1 linear velocity vector $\boldsymbol{v}$ as

$$\mathbf{v} = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} .$$

${}^{i-1}\mathbf{a}_i$ 6×1 spatial acceleration vector of in frame $\{i\}$ expressed in frame $\{i-1\}$. A spatial acceleration vector is composed by the 3×1 angular acceleration vector $\dot{\boldsymbol{\omega}}$ and the 3×1 linear acceleration vector $\dot{\boldsymbol{v}}$ as

$$\mathbf{a} = \begin{bmatrix} \dot{\boldsymbol{\omega}} \\ \dot{\boldsymbol{v}} \end{bmatrix} .$$

$\bar{\mathbf{I}}_i$ 3×3 rigid body inertia tensor of the i^{th} link with respect to its own coordinate system.

$\mathbf{I}_i$ 6×6 spatial inertia matrix for the i^{th} link, mapping velocity to momentum, and defined as

$$\mathbf{I}_i = \begin{bmatrix} \bar{\mathbf{I}}_i & \tilde{\mathbf{h}}_i \\ \tilde{\mathbf{h}}_i^{\mathsf{T}} & m_i \mathbf{1}_3 \end{bmatrix} , \tag{28}$$

where $\mathbf{h}_i = m_i \mathbf{s}_i$, m_i is the link's mass, $\mathbf{s}_i$ is the vector from the origin of the link's coordinate system to it's center of mass, and $\mathbf{1}_3$ is the 3×3 identity matrix.

$\mathbf{I}_i^\star$ 6×6 articulated body (AB) inertia matrix "felt" at the i^{th} link's coordinate system. The AB inertia represents the linear relationship between a force and an acceleration.

$\boldsymbol{\beta}_i^\star$ 6×1 vector containing the resultant bias force on the i^{th} link (including outward bias forces).

${}^i\mathbf{R}_{i-1}$ 3×3 rotational transformation matrix from frame $\{i-1\}$ to frame $\{i\}$.

${}^i\mathbf{X}_{i-1}$ 6×6 spatial transformation matrix from frame $\{i-1\}$ to frame $\{i\}$. It is defined as

$$ {}^i\mathbf{X}_{i-1} = \left[\begin{array}{c:c} {}^i\mathbf{R}_{i-1} & \mathbf{0} \\ \hdashline {}^i\mathbf{R}_{i-1}\,{}^{i-1}\tilde{\mathbf{p}}_i^{\mathrm{T}} & {}^i\mathbf{R}_{i-1} \end{array}\right] \tag{29} $$

σ_i Boolean parameter denoting the joint type. $\sigma_i = 1$ if joint i is a revolute joint and $\sigma_i = 0$ if it is a prismatic joint. $\bar{\sigma}_i$ denotes the negated value of σ_i.

$\dot{q}_i, \ddot{q}_i$ Relative velocity and relative acceleration between the inboard link (i) and the outboard link $(i-1)$.

$\boldsymbol{\phi}_i$ 6×1 spatial vector defining the joint motion axis (joint motion subspace) of joint i. If joint i is a revolute joint then $\boldsymbol{\phi}_i = [0,0,1,0,0,0]^{\mathrm{T}}$ and if joint i is a prismatic joint then $\boldsymbol{\phi}_i = [0,0,0,0,0,1]^{\mathrm{T}}$.

In order to simplify notation, the superscripts are dropped in case they are the same as the subscripts (e.g. ${}^{i-1}\mathbf{v}_{i-1} \rightarrow \mathbf{v}_{i-1}$).

Appendix B Transformations

This section lists all the transformations implemented in the simulation library. Since the notation used in this section is important, a clarifying example is first given before proceeding with the listing of different types of transformations.

□ EXAMPLE B.1 Let ${}^{\mathrm{A}}\mathbf{R}_{\mathrm{B}}$ be the rotation matrix that rotates frame $\{\mathrm{B}\}$ to frame $\{\mathrm{A}\}$. Also let ${}^{\mathrm{A}}\mathbf{x}$ represent a point in frame $\{\mathrm{A}\}$ and ${}^{\mathrm{B}}\mathbf{x}$ represent the same point in frame $\{\mathrm{B}\}$. Then

$$ {}^{\mathrm{A}}\mathbf{x} = {}^{\mathrm{A}}\mathbf{R}_{\mathrm{B}}\, {}^{\mathrm{B}}\mathbf{x} $$

transforms the vector ${}^{\mathrm{B}}\mathbf{x}$ from frame $\{\mathrm{B}\}$ to frame $\{\mathrm{A}\}$. Using this type of notation one may use the fact that the matrix subscript and the vector superscript "cancel". However, care must be taken not to rely on notation alone.

Since rotational matrices are othogonal, the following holds:

$$ {}^{\mathrm{A}}\mathbf{R}_{\mathrm{B}}^{-1} = {}^{\mathrm{A}}\mathbf{R}_{\mathrm{B}}^{\mathrm{T}} = {}^{\mathrm{B}}\mathbf{R}_{\mathrm{A}} $$

which are useful properties when reversing transformations. ◊

A rotation around the $\hat{Z}$-axis is represented with

$$ \mathbf{R}_z(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{30} $$

and a rotation around the $\hat{X}$-axis is represented by

$$ \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{31} $$

where θ is the angle of rotation. Note that the 2×2 sub-matrix, containing the trigonometric terms, defines a plane rotation.

For the sake of completeness, the matrix that produces a rotation in the XZ-plane is defined as

$$ \mathbf{R}_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{32} $$

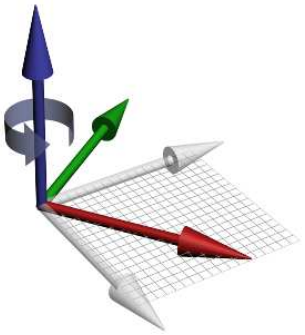


Figure 6: Rotation around the $\hat{Z}$-axis.

$$\mathbf{R}_z = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

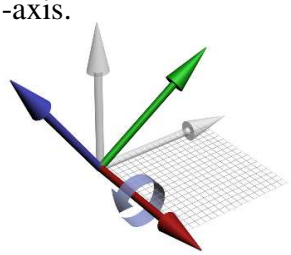


Figure 7: Rotation around the $\hat{X}$-axis.

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

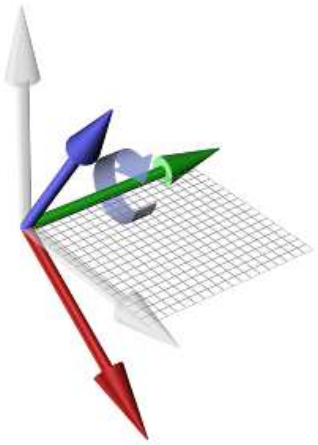


Figure 8: Rotation around the $\hat{Y}$-axis.

$$\mathbf{R}_y = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$

A general rotational transformation of a vector $\mathbf{p}$ can be seen as three successive planar rotations

$$\mathbf{p}' = \mathbf{R}_z(\gamma)\,\mathbf{R}_y(\beta)\,\mathbf{R}_x(\alpha)\,\mathbf{p}, \tag{33}$$

where the subscripts indicate the current axis of rotation and $\{\alpha, \beta, \gamma\}$ are the angles of rotation. In Eq. (33) the first rotation is about the $\hat{X}$-axis, the second about the $\hat{Y}$-axis, and the third about the $\hat{Z}$-axis. It is important to remember that the orientation of the two axes in the plane of rotation are also rotated. That is, any following rotations are taken about the "new" axis.

Reversing the transformation in Eq. (33) can be done using brute force linear algebra or by simply taking the the sequence of rotations in the reversed order (with the negative angles) as

$$\begin{aligned} \mathbf{p} &= \mathbf{R}_x(\alpha)^{-1}\mathbf{R}_y(\beta)^{-1}\mathbf{R}_z(\gamma)^{-1}\mathbf{p}' \\ &= \mathbf{R}_x(\alpha)^{\mathrm{T}}\mathbf{R}_y(\beta)^{\mathrm{T}}\mathbf{R}_z(\gamma)^{\mathrm{T}}\mathbf{p}' \\ &= \mathbf{R}_x(-\alpha)\mathbf{R}_y(-\beta)\mathbf{R}_z(-\gamma)\mathbf{p}' \,, \end{aligned} \tag{34}$$

where the orthogonality of the rotation matrix was used ($\mathbf{R}^{-1} = \mathbf{R}^{\mathrm{T}}$) in the middle equation above.

When using the MDH description for adjacent link frames, the rotational transformation from frame $\{i-1\}$ to frame $\{i\}$ is accomplished by applying two successive planar rotations; the first around the $\hat{X}$-axis and the second around the $\hat{Z}$-axis:

$${}^{i}\mathbf{R}_{i-1} = \mathbf{R}_z(\theta_i)\mathbf{R}_x(\alpha_i) = \begin{bmatrix} c\theta_i & s\theta_i & 0 \\ -s\theta_i & c\theta_i & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\alpha_i & s\alpha_i \\ 0 & -s\alpha_i & c\alpha_i \end{bmatrix}. \tag{35}$$

Going from frame $\{i-1\}$ to frame $\{i\}$ involves the following steps:

1. Rotation α_i around $\hat{X}_{i-1}$
2. Translation a_i along $\hat{X}_{i-1}$
3. Rotation θ_i around (the new) $\hat{Z}_i$
4. Translation d_i along (the new) $\hat{Z}_i$

The vector from the origin of frame $\{i-1\}$ to frame $\{i\}$ is defined as

$${}^{i-1}\mathbf{p}_i = [a_i, 0, 0]^{\mathrm{T}} + \mathbf{R}_x(-\alpha_i)[0, 0, d_i]^{\mathrm{T}} = \begin{bmatrix} a_i \\ -d_i \sin(\alpha_i) \\ d_i \cos(\alpha_i) \end{bmatrix}, \tag{36}$$

where the vector containing d_i was rotated from frame $\{i\}$ to frame $\{i-1\}$. The superscript indicates that the position vector is expressed with respect to frame

$\{i-1\}$. In order to shorten the notation in the description of the AB-algorithm, spatial vectors are used to group together properties as velocities, accelerations, forces. For instance, the spatial velocity vector is defined as $\mathbf{v} = \begin{bmatrix} \boldsymbol{\omega}^{\mathrm{T}} & \boldsymbol{v}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}$. Hence, a spatial, 6×1 vector is formed by concatenating two cartesian vectors.

An outward transformation of a spatial vector is performed by applying the following transformation matrix

$$ {}^{i}\mathbf{X}_{i-1} = \left[\begin{array}{c:c} {}^{i}\mathbf{R}_{i-1} & \mathbf{0} \\ \hdashline {}^{i}\mathbf{R}_{i-1}\ {}^{i-1}\tilde{\mathbf{p}}_{i}^{\mathrm{T}} & {}^{i}\mathbf{R}_{i-1} \end{array} \right], \tag{37} $$

where ${}^{i-1}\tilde{\mathbf{p}}_i$ is the position vector from frame $\{i-1\}$ to frame $\{i\}$ (w.r.t. frame $\{i-1\}$), composed such that $\tilde{\mathbf{p}}\boldsymbol{\omega} = \mathbf{p} \times \boldsymbol{\omega}$. The skew-symmetric matrix $\tilde{\mathbf{p}}$ is taken as its transpose in order to compensate for a change of sign which arises when taking the cross-product between two vectors in the reversed order ($\tilde{\mathbf{p}}^{\mathrm{T}}\boldsymbol{\omega} = -\mathbf{p} \times \boldsymbol{\omega} = \boldsymbol{\omega} \times \mathbf{p}$). Inertial quantities can also be grouped together, much in the same sense as in the case of spatial vectors. The difference lies in the fact that cartesian matrices instead of cartesian vectors are grouped together.

B.1 General transformations

Since the AB algorithm is iterative, transformations from inboard links to outboard links (and vice versa) constitute a major part of the algorithm. From a computational point of view, it is important to implement all transformations as efficiently as possible.

General transformations between frames need six parameters. By using joints which can be characterized by a single axis, it is possible to reduce the number of parameters to only four. The MDH scheme uses the fact that any two lines in space have a common perpendicular. Generally, this perpendicular is unique but there are situations when this is not the case [4]. In the MDH scheme the $\hat{Z}$-axis (of a link frame) is aligned with the joint axis and the $\hat{X}$-axis is aligned with the common perpendicular between adjacent joint axes.

Below follows a listing of all transformations in their general form. Section B.2 lists computationally more efficient transformations for links with only a single degree of freedom.

General outboard rotational transformation Transformation of a vector in frame $\{i-1\}$ to the outboard frame $\{i\}$ is done by

$$ {}^{i}\boldsymbol{\omega}_{i-1} = {}^{i}\mathbf{R}_{i-1}\ \boldsymbol{\omega}_{i-1}. \tag{38} $$

General inboard rotational transformation Transformation of a vector in frame $\{i\}$ to the inboard frame $\{i-1\}$ is done by

$$^{i-1}\boldsymbol{f}_i = {}^{i-1}\mathbf{R}_i^{\mathrm{T}}\, \boldsymbol{f}_i = {}^{i}\mathbf{R}_{i-1}\, \boldsymbol{f}_i. \tag{39}$$

General outboard spatial transformation Transformation of a 6×1 spatial vector in frame $\{i-1\}$ to the outboard frame $\{i\}$ is done by

$$^{i}\mathbf{v}_{i-1} = {}^{i}\mathbf{X}_{i-1}\, \mathbf{v}_{i-1}. \tag{40}$$

General inboard spatial transformation Transformation of a 6×1 spatial vector in frame $\{i\}$ to the inboard frame $\{i-1\}$ is done by

$$^{i-1}\mathbf{f}_i = {}^{i-1}\mathbf{X}_i^{\mathrm{T}}\, \mathbf{f}_i = {}^{i}\mathbf{X}_{i-1}\, \mathbf{f}_i. \tag{41}$$

Rotational congruence transformation to inboard frame Transformation of a 3×3 symmetric matrix in frame $\{i\}$ to the inboard frame $\{i-1\}$ is done by the following congruence transformation

$$^{i-1}\bar{\mathbf{I}}_i = {}^{i}\mathbf{R}_{i-1}^{\mathrm{T}}\, \bar{\mathbf{I}}_i\, {}^{i}\mathbf{R}_{i-1}. \tag{42}$$

where it is possible to use the fact that $\bar{\mathbf{I}}$ is symmetric to reduce the computational cost.

Rotational congruence transformation to inboard frame Transformation of a general 3×3 matrix in frame $\{i\}$ to the inboard frame $\{i-1\}$ is done by the following congruence transformation (same as above)

$$^{i-1}\mathbf{G}_i = {}^{i}\mathbf{R}_{i-1}^{\mathrm{T}}\, \mathbf{G}_i\, {}^{i}\mathbf{R}_{i-1}. \tag{43}$$

Spatial congruence transformation to inboard frame Transformation of a 6×6 spatial matrix in frame $\{i\}$ to the inboard frame $\{i-1\}$ is done by the following congruence transformation

$$^{i-1}\mathbf{N}_i = {}^{i}\mathbf{X}_{i-1}^{\mathrm{T}}\, \mathbf{N}_i\, {}^{i}\mathbf{X}_{i-1}. \tag{44}$$

B.2 Link transformations

This section describes link-to-link transformations for links with a single degree of freedom. These transformations are more efficient than using the general transformations as defined in the previous section.

Rotational transformation from inboard frame Rotation of a 3×1 cartesian vector in frame $\{i-1\}$ to the outboard frame $\{i\}$ is done by applying two successive planar rotations as

$$\mathbf{p}_i = {}^i\mathbf{R}_{i-1}\,\mathbf{p}_{i-1} = \mathbf{R}_z(\theta_i)\,\mathbf{R}_x(\alpha_i)\,\mathbf{p}_{i-1}, \tag{45}$$

where α_i and θ_i are two of the MDH parameters for the i^{th} link, as defined with respect to link $(i-1)$.

Rotational transformation to inboard frame Rotation of a 3×1 cartesian vector in frame $\{i\}$ to the inboard frame $\{i-1\}$ is done by

$$\mathbf{p}_{i-1} = \mathbf{R}_x^{\mathrm{T}}(\alpha_i)\,\mathbf{R}_z^{\mathrm{T}}(\theta_i)\,\mathbf{p}_i\ . \tag{46}$$

Spatial transformation from inboard frame Transformation of a 6×1 spatial vector in frame $\{i-1\}$ to the outboard frame $\{i\}$ is most efficiently implemented by applying two successive planar screw transformations as

$$\mathbf{v}_i = \mathbf{X}_z(d_i, \theta_i)\,\mathbf{X}_x(a_i, \alpha_i)\,\mathbf{v}_{i-1} \tag{47}$$

where the spatial planar screw transformations are defined as

$$\mathbf{X}_x(a_i, \alpha_i) = \left[\begin{array}{c:c} \mathbf{R}_x(\alpha_i) & \mathbf{0} \\ \hdashline \mathbf{R}_x(\alpha_i)\ \tilde{\mathbf{p}}_x^{\mathrm{T}} & \mathbf{R}_x(\alpha_i) \end{array}\right] \tag{48}$$

and

$$\mathbf{X}_z(d_i, \theta_i) = \left[\begin{array}{c:c} \mathbf{R}_z(\theta_i) & \mathbf{0} \\ \hdashline \mathbf{R}_z(\theta_i)\ \tilde{\mathbf{p}}_z^{\mathrm{T}} & \mathbf{R}_z(\theta_i) \end{array}\right], \tag{49}$$

where $\tilde{\mathbf{p}}_x^{\mathrm{T}} = [a_i, 0, 0]^{\mathrm{T}}$ and $\tilde{\mathbf{p}}_z^{\mathrm{T}} = [0, 0, d_i]^{\mathrm{T}}$.

Spatial coordinate transformations cannot be handled the same way as cartesian coordinate transformations. Letting $\mathbf{X}$ be any spatial transformation matrix, it is not possible to reverse that transformation by simply taking the matrix transpose, $\mathbf{X}^{\mathrm{T}}$, as in the cartesian case ($\mathbf{R}^{-1} = \mathbf{R}^{\mathrm{T}}$). Using the inverse of the transformation matrix is of course mathematically correct but inefficient from a computaional point of view. Instead, one can use the *spatial transpose*. This is possible due to the fact that coordinate transformations are spatially orthogonal (see [5] for more details).
The spatial transpose operator is defined as

$$\mathbf{M}^{\mathrm{S}} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{\mathrm{S}} = \begin{bmatrix} \mathbf{D}^{\mathrm{T}} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{C}^{\mathrm{T}} & \mathbf{A}^{\mathrm{T}} \end{bmatrix}\ , \tag{50}$$

where $\{\mathbf{A,B,C,D}\} \in \mathbb{R}^{3\times 3}$ are 3×3 block matrices. In the case of spatial coordinate transformations, the diagonal blocks are identical (**A=D**) and the inverse of a spatial coordinate transformation can easily be constructed by assembling the transposed blocks and use the relation $\mathbf{X}^{-1} = \mathbf{X}^{\mathrm{S}}$.

Another way of reversing a spatial coordinate transformation consisting of successive planar screws is to reverse the sequence of the screw transformations and change the sign of their arguments as

$$\mathbf{X}^{-1} = \mathbf{X}_x(-a, -\alpha)\, \mathbf{X}_z(-d, -\theta) \tag{51}$$

which is equivalent to (using the spatial transpose)

$$\mathbf{X}^{-1} = \mathbf{X}_x^{\mathrm{S}}(a, \alpha)\, \mathbf{X}_z^{\mathrm{S}}(d, \theta) \quad . \tag{52}$$

Spatial transformation to inboard frame Transformation of a 6×1 spatial vector in frame $\{i\}$ to the inboard frame $\{i-1\}$ is most efficiently implemented by applying two successive planar screw transformations as

$$\mathbf{v}_{i-1} = \mathbf{X}_x(-a_i, -\alpha_i)\, \mathbf{X}_z(-d_i, -\theta_i)\, \mathbf{v}_i \tag{53}$$

Appendix C Delphi and object-oriented programming

Unlike procedural programming, object-oriented programming (OOP) provides a higher level of abstraction. At its simplest level, OOP can be seen as a way of modularizing the code, increasing the readability, and making the code easier to maintain. OOP techniques provide the means of organizing the code.

The basic concept in OOP is the use of classes. A class is a collection of variables containing methods for operating on those variables. An object is an instance of a class. Keywords in OOP are: (1) encapsulation, (2) inheritance, and (3) polymorphism.

Encapsulation concerns the way data can be made private to the class, i.e. only the class itself is allowed to modify its data. Access to parts of a class is done through the interface which is part of the definition of the class. In general, there are basically three levels of visibilty (access levels) for both fields (data) and methods (functions and procedures) in the class definition:

- **Private** elements are only visible from within the class itself.
- **Protected** elements extends the private visibility to include child classes, i.e. it defines the interface to its descendants.
- **Public** elements defines the interface to the outside. Public elements unrestricted access.

Inheritance means that descendant classes inherits all the fields and methods from its ancestor (base class). It can be seen as a way of extending (or specializing) an already defined class (see Listing 4).

Polymorphism is tightly coupled to inheritance and run-time behavior; without inheritance, polymorphism would not be possible. Polymorphism is defined as the ability of related (but different) objects to implement the same method in their own way. For example, a circle and a square object each have a Paint method. From the outside, these methods look the same but are both implemented in different ways. The ability of an object to implement a different behavior during run-time is referred to as *late binding*.

Listing 4 shows an example of an abstract base class and a derived, concrete class as defined in Delphi. The class `TBaseAbstract` is called abstract since it does not provide an implementation of the `Paint` procedure. It is the responsibility of the derived class `TConcrete` to implement that procedure. An example on how to use the `TConcrete` class is as follows

```
var
  obj: TConcrete;
begin
```

```
  obj := TConcrete.Create;
  obj.Paint;
  obj.Free;
end;
```

where an instance of the class is created (the object `obj`) by calling the default constructor `Create`, then the `Paint` procedure is called, and finally the object is destroyed by calling the default destructor `Free`.

Listing 4: A simple example of an abstract base class and a derived class.

```
interface
type
  TBaseAbstract = class
  private
    data: integer;
  public
    procedure Paint; virtual; abstract;
  end;

  TConcrete = class(TBaseAbstract)
  private
    new_data: integer;
  public
    procedure Paint; override;
  end;

implementation
procedure TConcrete.Paint;
begin
  // implementation of Paint
end;
```

References

[1] R. C. ARKIN, *Behavior-based robotics*, The MIT Press, 1998.

[2] BORLAND, *Delphi*. `http://www.borland.com/delphi`.

[3] R. BROOKS, *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, RA-2 (1986), pp. 14–23.

[4] J. J. CRAIG, *Introduction to Robotics: Mechanics and Control*, Addison-Wesley Publishing Company, 2nd ed., 1989.

[5] R. FEATHERSTONE, *Robot Dynamics Algorithms*, Kluwer Academic Publishers, 1987.

[6] J. HOLLAND, *Adaptation in natural and artificial systems*, MIT Press, Cambridge, MA, 1992.

[7] O. KHATIB, *Real-time obstacle avoidance for manipulators and mobile robots*, in Proceedings of the IEEE International Conference on Robotics and Automation, 1985, pp. 500–505.

[8] J. R. KOZA, *Gentetic Programming: On the programming of Computers by Natural Selection*, MIT Press, Cambridge, MA, 1992.

[9] P. MAES, *The dynamics of action selection*, in Proc. of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89), 1989, pp. 991–997.

[10] D. MCFARLAND, *Animal behavior: Psychobiology, Ethology and Evolution*, Prentice Hall, 3rd edition ed., 1998.

[11] S. MCMILLAN, *Computational Dynamics for Robotic Systems on Land and under Water*, PhD thesis, The Ohio State University, Columbus, OH, Summer 1994.

[12] M. MITCHELL, *An introduction to genetic algorithms*, The MIT Press, 1996.

[13] H. MORAVEC, *Robot: Mere machine to transcendent mind*, Oxford University Press, 1999.

[14] P. PIRJANIAN, *Behavior-coordination mechanism – state-of-the-art*, Technical report IRIS-99-375, Institute for Robotics and Intelligent Systems, University of Southern California, October 1999.

[15] P. PIRJANIAN AND M. MATARIC, *A decision-theoretic approach to fuzzy behavior coordination*, in Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation, Monterey, CA, November 1999.

[16] W. PRESS, *Numerical recipes in C the art of scientific computing*, Cambridge University Press, 1992.

[17] I. RECHENBERG, *Evolutionsstrategien*, Holtzmann-Froboog, 1994.

[18] J. ROSENBLATT, *Damn: A distributed architecture for mobile navigation*, in AAAI 1995 Spring Symposium on lessons learned for implemented software architectures for physical agents, March 1995, pp. 167–178.

[19] M. WAHDE, *A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions*, Journal of Systems and Control Engineering, 217 (2003), pp. 249–258.

[20] M. W. WALKER AND D. E. ORIN, *Efficient dynamic computer simulation of robotic mechanisms*, Journal of Dynamic Systems, Measurement, and Control, 104 (1982), pp. 205–211.

[21] K. WOLFF AND P. NORDIN, *Evolution of efficient gait with humanoids using visual feedback*, in Proceedings of the IEEE-RAS International Conference on Humanoid Robots, 2001, pp. 99–106.

Paper III

Application of the utility function method for behavioral organization in a locomotion task

in

IEEE Transactions on Evolutionary Computation, Volume 9, Issue 5, October 2005, pp. 506–521.

Application of the Utility Function Method for Behavioral Organization in a Locomotion Task

Jimmy Pettersson and Mattias Wahde

Abstract—**The generation of a complete robotic brain for locomotion based on the utility function (UF) method for behavioral organization is demonstrated. A simulated, single-legged hopping robot is considered, and a two-stage process is used for generating the robotic brain. First, individual behaviors are constructed through artificial evolution of recurrent neural networks (RNNs). Thereafter, a behavioral organizer is generated through evolutionary optimization of utility functions.**

Two systems are considered: a simplified model with trivial dynamics, as well as a model using full newtonian dynamics.

In both cases, the UF method was able to generate an adequate behavioral organizer, which allowed the robot to perform its primary task of moving through an arena, while avoiding collisions with obstacles and keeping the batteries sufficiently charged.

The results for the simplified model were better than those for the dynamical model, a fact that could be attributed to the poor performance of the individual behaviors (implemented as RNNs) during extended operation.

Index Terms—**Behavioral organization, behavior-based robotics, evolutionary algorithms (EAs), structural optimization, utility functions (UFs).**

I. Introduction and Motivation

DURING THE last few years, the field of autonomous robotics has been growing at a rapid rate. While the results have often been impressive, several challenges remain on the road to truly intelligent autonomous machines. Two such challenges, which will be addressed in this paper, concern locomotion of legged systems and behavioral organization in behavior-based robots, i.e., the process of activating the appropriate behavior at all times.

The first topic, legged locomotion, has been considered by many authors [1]–[4], and many implementations exist in bipedal robots such as Asimo, Qrio, etc. (see, e.g., [5] for a review of bipedal robotics research). Several methods have been proposed for posture control and locomotion of bipedal robots. The most common methods for generating stable gaits include the use of posture controllers together with real-time trajectory planners based on the *zero moment point* (ZMP) [6], [7]. Using explicit reference trajectories is, however, not suitable for robots that operate in uncontrolled, dynamic environments. In such environments, the robot must be able to handle unforeseen events such as obstacles that suddenly appear near the robot or actual physical perturbations. Thus, what is required is a method for *walking by sensing*, where the robot makes its own decisions about what action to take based on sensory input. In other words, the robot must be equipped not only with behaviors for locomotion and collision avoidance, but also with a means of activating the right behavior at the right time.

Thus, the second topic, behavioral organization,[1] constitutes one of the most important (and also most difficult) problems in behavior-based robotics (see, e.g., [8]–[10], and references therein), and many different methods have been suggested for solving this problem. In general, most methods require a significant amount of hand-coding and manual fine-tuning of the behavioral organization system [8], [10]. Here, the problem will be approached using the recently developed utility function (UF) method [10], in which the behavioral organization is obtained using an evolutionary algorithm (EA), rather than through hand-coding.

While the UF method itself uses an EA to generate behavioral organization, the constituent behaviors can be generated by any method. However, since legged robots are modeled on biological organisms, the use of biologically inspired computation methods, such as EAs applied to recurrent neural networks (RNNs) is a logical strategy for the generation of behaviors [11], and also the method of choice for this paper.

In this paper, it is shown how a complete (albeit rather simple) robotic brain[2] for locomotion can be evolved, starting with the evolution of constituent behaviors and completing the task by evolving the behavioral organizer, thus avoiding any manual fine-tuning of the robotic brain. As the emphasis lies on the generation of the robotic brain, and in particular its behavioral organization mechanisms, a simplified (compared to a bipedal robot) legged system has been used, namely a one-legged, hopping robot. Unlike, say, a wheeled robot, any legged system (whether monopedal or bipedal) cannot, for example, stop abruptly without losing its balance. Thus, while considerably simpler than a bipedal robot, robust motion of a one-legged robot is also a challenging task, and certainly sufficiently challenging to make behavioral organization far from trivial.

II. Problem Description

The aim of this paper is to provide a simple one-legged robot with a complete robotic brain for simple (linear) locomotion,

Manuscript received September 23, 2003; revised September 1, 2004.

The authors are with the Department of Applied Mechanics, Chalmers University of Technology, 412 96 Göteborg, Sweden (e-mail: jimmy.pettersson@me.chalmers.se; mattias.wahde@me.chalmers.se).

Digital Object Identifier 10.1109/TEVC.2005.850262

[1]Behavioral organization is sometimes also referred to as behavioral selection or action selection. Here, the term behavioral organization will be used throughout.

[2]Throughout this paper, the term *robotic brain* will be used instead of the more limited term *control system*, to emphasize that the generated behaviors are ultimately to be added as components in a complete behavior-based robotic brain, incorporating not only motor behaviors but other (e.g., cognitive) behaviors as well.

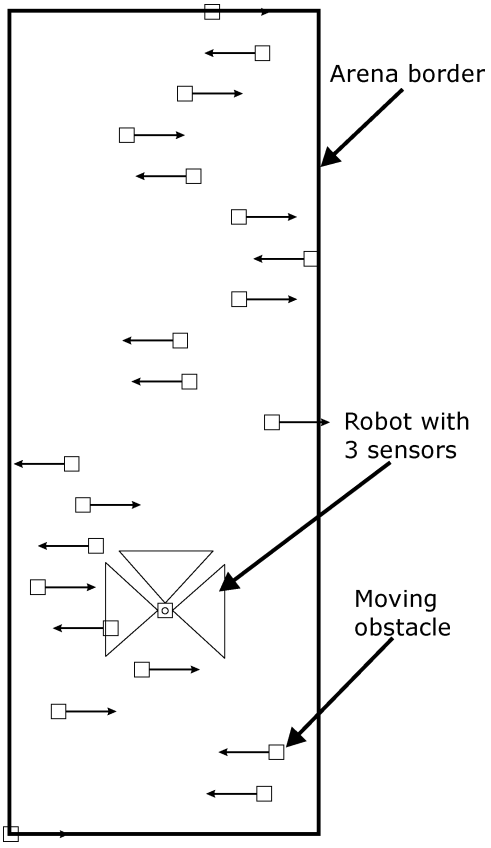


Fig. 1. The robot in its environment. The goal of the robot is to move across the arena without colliding with any obstacle. Periodic boundary conditions are used in the horizontal direction. The length of the arena is 8 m, and its width is 3 m.

including the ability to avoid collisions with moving obstacles. The problem faced by the robot is illustrated in Fig. 1. Its goal is to move from bottom to top in the arena shown (from above) in the figure, without colliding with any of the horizontally moving obstacles, and without running out of battery energy.

While seemingly simple, the problem is in fact quite complex. First of all, a dynamically modeled one-legged robot, as the one used here, cannot start (or stop) instantaneously. Instead, it needs to build up (or wind down) its speed by moving its single leg back and forth. Second, the robot must occasionally stop in order to charge its batteries, and must thus select the appropriate time (and place) for doing so. Third, the robot's sensors have a limited range. Finally, the obstacles, moving in different directions, may sometimes force the robot to move backward, even though it is only rewarded (at the end of an evaluation) for moving forward.

As a minimum, such a robot must be equipped with four behaviors: *move forward*, *move backward*, *stop* (i.e., lower the speed to zero), and *charge batteries* (by remaining stationary, assuming that the charging is performed using, e.g., solar cells). Equally important, as indicated by the previous paragraph, is that the robot be equipped with a method for behavioral organization, i.e., for selecting and activating appropriate behaviors in any given situation. Here, the UF method for behavioral organization (see [10] and Section III-B1) will be used for this purpose.

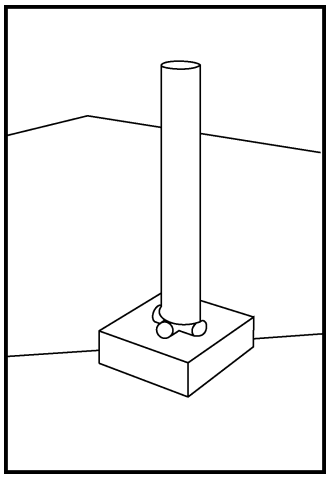


Fig. 2. Simulated one-legged robot. The leg is actuated by two revolute joints, located at the center of the supporting plate (foot). Four contact points under the foot provide information about the external forces acting on the robot.

In order for behaviors to be organized, however, they must first be generated. Here, the constituent behaviors will be represented by recurrent, continuous-time neural networks (RNNs), and the generation of these behaviors, also by means of artificial evolution, will be studied as open-ended problems, where the complexity of the evolving networks is allowed to change (increase or decrease in size). In addition, several different sets of input signals will be considered, in order to investigate whether other signals than the standard ones (joint positions and angular velocities) can provide information that is more suitable as input to an RNN.

III. Models and Methods

In this section, the simulated system will be described, as well as the methods used for generating and organizing behaviors.

A. Simulated System

The simulated system, shown in Fig. 2, consists of two connected rigid bodies, a rod (leg) with mass m_2 and a plate (foot) with mass m_1. Relative motion between the two rigid bodies is allowed via two revolute joints (cf. ankle joints). In addition to the two degrees of freedom (DOF) at the joints, the entire system is free to move in space, thus adding six DOF, giving a total of eight DOF. However, only the two DOF in the ankle joint are actuated.

1) Motor Model: The two revolute joints between the foot and the leg are driven by DC motors models. Motor parameters were set according to a specification of the motor DC-Micromotors Series 3863, manufactured by MicroMo [12] and commercially available through the Faulhaber Group [13].

The model consists of a source voltage V, armature resistance R, and the back EMF (V_e) generated by the motor as

$$V = Ri + V_e \tag{1}$$

where i is the current through the circuit. The source voltages of the two DC motors are set by the two outputs from the RNN (after being scaled to the appropriate voltage range).

Since the back EMF is proportional to the angular velocity of the motor $V_e = K_e\omega$, and the torque produced is proportional to the current $T_0 = K_t\, i$, the final torque developed by the motor is

$$T = \frac{K_t}{R}(V - K_e\omega) - T_f \tag{2}$$

where T_f is an opposing torque due to friction in the motor.

In addition, a lossless gearbox is used to scale the output from the DC motor. Since the gearbox is lossless, equal power is generated on both sides of the gearbox. As such, the gearbox effectively functions as a simple scaling unit, having the following relationships:

$$T_l = GT \tag{3}$$

$$\omega = G\omega_l \tag{4}$$

where T_l is the torque applied to the rigid body, ω_l is the angular velocity of the rigid body, and G is the gearbox ratio. Also, the (absolute) maximum torque allowed by the gearbox was set to $\tau_{\max}$.

In real systems, it is difficult (or very expensive) to achieve the same updating frequency as in simulated systems. This is mainly due to limitations in single components or in combined hardware. Therefore, the motor model used in the simulations is only allowed to update with a frequency of 100 Hz, even though the step size used when integrating the equations of motion is smaller (0.002 s).

2) Sensors: In traditional control theory, the joint angles and joint angular derivatives would be chosen as input signals to the system. However, angles and angular derivatives are not explicitly available for the balancing of biological organisms (even though some organisms are able to estimate joint angles using vision or using sensors in muscles and joints). Instead, the signals available are, e.g., the pressure distributions under the feet, as well as the information from the balancing organ (essentially, an accelerometer). Thus, when approaching the problem of balancing using biologically inspired computation methods, it is appropriate at least to consider the *possibility* of using input signals other than the traditional ones.

Thus, in addition to the joint angles (φ_1, φ_2) and joint angular derivatives ($\dot{\varphi}_1$, $\dot{\varphi}_2$), which will be referred to as *classical (input) signals*, four contact forces (F_1, F_2, F_3, F_4), measured at the corners of the plate representing the foot, and the accelerations of the center of mass of the leg ($\bar{a}_x$, $\bar{a}_y$, $\bar{a}_z$) were added as potential input signals to the system. The four contact forces are obtained through massless spring and damper systems, as shown in Fig. 3. In essence, the four contact sensors measure the pressure distribution under the foot. Since the foot is rigid and is assumed to be located on a flat surface, four forces suffice for this purpose.

The accelerations represent the signals generated by a balancing organ. Note that the accelerations are smoothed (to reduce noise) using a moving average before being presented to the balancing system. The seven added potential input signals will be referred to as *biological (input) signals*.

The robot was also equipped with three simple proximity sensors (s_1, s_2, and s_3), responsible for detecting obstacles in the environment. Note that these sensors, however, were *not* made available to the robot when individual behaviors were generated. Proximity sensors were *only* used in simulations involving behavioral organization.

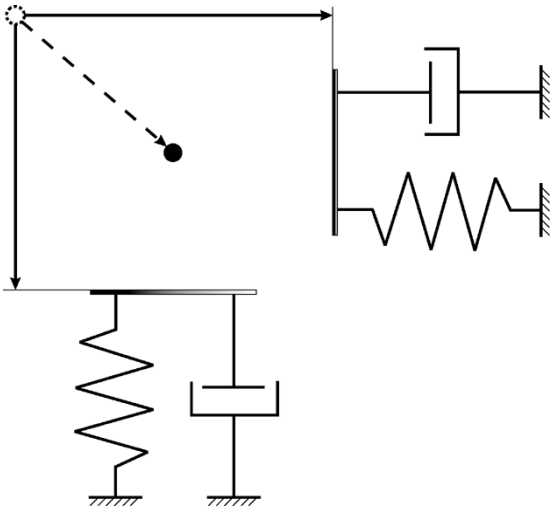


Fig. 3. Spring and damper models for a single contact point. The two models generate the horizontal and vertical force components, respectively. The white circle represents the initial contact point.

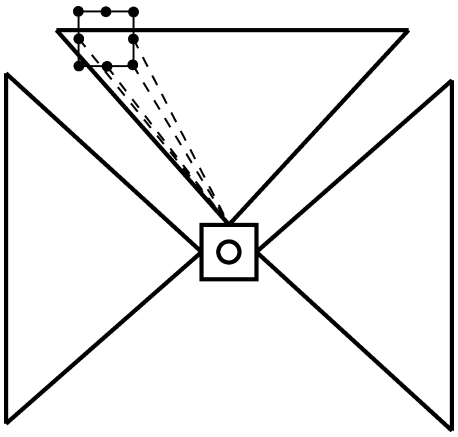


Fig. 4. The robot as seen from above, equipped with three simple proximity sensors, each with a triangular-shaped detection range. The filled black circles illustrates points on an obstacle that are detectable by the range sensors. Dashed lines indicates detected points in this particular configuration.

One proximity sensor is placed in the robot's forward direction and the other two are placed on the right and on the left sides of the robot (see Fig. 4). As shown in the picture, each sensor has a triangular detection range, with the base s_b and height s_h. In case an obstacle is detected by a sensor, the output of the sensor is calculated, somewhat arbitrarily, as

$$S_i(d) = e^{-2\frac{d}{d_{\max}}} \tag{5}$$

where d is the distance to the detected obstacle and $d_{\max}$ is the maximum range of the sensor.

From a sensor's point of view, each obstacle is composed of eight points; the four corner points and the four midpoints in between. During simulation, each sensor goes through all obstacles and checks for point inclusion in its range-defining triangle. If multiple points are within sensor range (see Fig. 4), the sensor output is calculated using the distance to the closest point. For

simplicity (and to speed up the simulations), only coordinates in the ground plane are considered in the detection of obstacles.

In addition to the three simple proximity sensors, a fourth (internal) sensor is available for monitoring of battery level (E), which has maximum capacity $E_{\max}$. Like the proximity sensors, this sensor was *only* used in the simulations involving behavioral organization.

The robot's energy usage is based on the torques generated by the motors (τ_1, τ_2) and the angular velocity of each joint ($\dot{\varphi}_1$, $\dot{\varphi}_2$), with the instantaneous energy usage calculated as

$$\Delta E = -\Delta t \left(|\tau_1 \dot{\varphi}_1| + |\tau_2 \dot{\varphi}_2| \right) \tag{6}$$

where Δt is the step size used in the integration of the equations of motion.

In case the robot charges its batteries (which requires activation of the charging behavior, see Section IV-A), the energy in the battery changes as

$$\Delta E = \begin{cases} 0 \text{ J/s}, & \text{(for the first } T_0 \text{ seconds)} \\ \Delta E_0 \text{ J/s}, & \text{(thereafter)} \end{cases}. \tag{7}$$

Thus, in order to gain energy the robot must keep the charging behavior active for more than T_0 s.

It should be noted that, during simulations, sensors are only allowed to update with a frequency of 100 Hz (for the same reasons mentioned earlier in Section III-A1).

3) Simplified System: In addition to the dynamically modeled one-legged robot, a simplified system (with trivial dynamics) was used as well, mainly for verification that the settings used for the UF method (see Section III-B1) were chosen correctly. During simulations with this simplified system, the four behaviors were implemented by simply setting the velocity of the robot directly (to prespecified values) instead of integrating the equations of motion. For the simplified system, the speed in the *move forward* behavior is set to v_F, and the speed in the *move backward* behavior is set to v_B.

4) Parameter Settings for the Robot: The parameter settings, i.e., the actual values used for the constants introduced in this section, are specified in Table I.

TABLE I
PARAMETER SETTINGS FOR THE SIMULATED ROBOT. THE TWO FINAL PARAMETERS REFER TO THE SIMPLIFIED SYSTEM (SEE SECTION III-A3)

Parameter	Value	Description
m_1	1.0 kg	Foot mass
m_2	0.5 kg	Leg mass
E_{max}	$50 - 500$ J	Maximum battery level
V	± 24 V	DC motor voltage range
K_t	0.0333 Nm/A	Torque constant
K_e	0.0333 Ns/rad	Back EMF constant
R	0.62 Ω	DC motor armature resistance
G	4.8	Gearbox ratio
τ_{max}	10 Nm	Maximum torque
ΔE_0	10 J/s	Battery charge rate
T_0	0.5 s	Battery charge delay
s_b	0.9 m	Base length of sensor triangle
s_h	0.5 m	Height of sensor triangle
v_F	0.15 m/s	Forward velocity (simplified syst.)
v_B	0.12 m/s	Backward velocity (simplified syst.)

B. Robotic Brain

1) Behavioral Organizer: In this paper, the behavioral organizer, i.e., the system for selecting which behavior to activate in any given situation, is obtained by means of the UF method [10]. This ethologically inspired behavioral organization method uses artificial evolution to generate the behavioral organizer, and has the advantage of requiring only a minimum of hand-coding by the user. An additional advantage is the *possibility* (not the *requirement*) of including prior knowledge before the evolutionary optimization is initiated.

Here, only a brief introduction to the method will be given. For a more complete description, see [10]. In the UF method, each available behavior B_i is associated with a utility function U_i that depends on the state variables of the robot, such as sensor readings (e.g., contact forces and joint angles), battery energy levels, etc. Once the utility functions have been generated, behavioral organization is simple: at any given time, the behavior with the largest utility value will be active. The problem, of course, is to generate the utility functions. In the UF method, these functions are obtained via artificial evolution. The user need only provide an ansatz for each utility function, e.g., an pth-degree polynomial in the state variables, for some positive integer p. Thus, when using the UF method, a population of behavioral organizers is generated, with initially random polynomial coefficients in their utility functions. Each behavioral organizer is evaluated, by allowing it to control the robot for a certain period of time, after which a fitness value is assigned based on the overall performance of the robot, and evolution can proceed as usual, until the best behavioral organizer has been found.

In general, assigning a suitable fitness measure is difficult for a task involving several behaviors. Thus, another advantage with the UF method is that fitness functions need only be provided for the *task behaviors* of the robot, i.e., the behaviors directly concerned with the task of the robot. For other (*auxiliary*) behaviors, no fitness functions need be assigned.

As an example, consider a simple exploration robot equipped with two behaviors: one task behavior that simply keeps the robot moving in a random fashion, and one auxiliary behavior responsible for charging the batteries (by standing still, assuming the robot is equipped with solar panels). Now, the charging behavior is clearly needed in most cases, but preferably the user should not have to assign a fitness function to it, as this would require making an assessment, for each situation, of the relative importance of the two behaviors. By contrast, in the UF method, only the task behavior is assigned a fitness function (e.g., distance moved). The occasional activation of the charging behavior is the task of the behavioral organizer. If properly evolved, the two utility functions will be such that the utility of battery charging will rise as the energy in the battery falls, so that, eventually, the charging behavior will be activated for a while, thus allowing the robot (after charging has been completed and the utility of charging falls below the utility

of the task behavior) to continue with its task behavior and thereby achieving higher fitness. If instead charging had not been activated, the robot would of course come to a standstill after emptying its batteries, and would not be able to increase its fitness. Thus, there is a strong incentive for the EA to find utility functions such that the charging behavior is sometimes activated.

While this example is strongly simplified (in most cases, several additional auxiliary behaviors will be needed, e.g., obstacle avoidance, etc.), it illustrates the principle behind the UF method.

Furthermore, in addition to the obvious state variables, the UF method introduces so called *internal abstract variables*. These are variables without a direct physical counterpart, which are used, e.g., for avoiding rapid switching between behaviors [10]. In a biological analogy, these variables can be seen as representing hormone levels, which can act as internal variables governing the selection of behaviors.

Finally, the UF method also introduces the notion of *behavior time*, i.e., a behavior-specific time variable t_i which measures the time since behavior i was last activated, and which is equal to zero when behavior i is *not* active.

2) Constituent Behaviors: For the implementation of behaviors, there are many architectures available, such as, e.g., finite-state machines [14], fuzzy logic controllers [15], and neural networks [16]. In addition, the components of classical control systems, such as PD controllers, etc., are also applicable for certain low-level, noncognitive behaviors, such as posture control in the absence of strong perturbations. In this paper, behaviors will be implemented in the form of RNNs, which are described next.

Of course, other architectures than RNNs could, in principle, have been used for the implementation of behaviors. However, neural networks are generally able to represent and process a data flow regardless of its structure (or lack thereof). In this study, several different sets of input signals are used. From these often disparate sets of signals (such as, e.g., angular positions together with contact forces), useful information must be extracted, making neural networks a natural choice.

In this paper, the robotic brain contains four different behaviors, implemented as RNNs. It should be stressed here that the evolution of the constituent behaviors is performed completely independently of the process of behavioral organization. Thus, when the four behaviors are evolved, they are considered as separate entities, and no attempt is made, e.g., to make them particularly suitable for the subsequent behavioral organization. Thus, the proximity sensors and the battery energy sensor (see Section III-A2) were *not* used when evolving the constituent behaviors.

The fitness functions used for evolving each of the four behaviors should *not* be confused with the fitness function used by the UF method. The latter is a function for selecting between behaviors that are already present, regardless of how they were obtained (e.g., by means of artificial evolution or some other method).

The general architecture used for the four constituent behaviors will now be described.

a) Recurrent neural networks: The RNNs used here consist of a set of input elements that send the input signals to the neurons i, whose dynamics is given by

$$\tau_i \dot{x}_i + x_i = \sigma\left(b_i + \sum_{j=1}^{n} w_{ij} x_j + \sum_{j=1}^{m} w_{ij}^{\mathrm{IN}} y_j\right) \tag{8}$$

where i runs from 1 to n (the number of neurons), and m is the number of input elements. Here, input elements refer to a set of the available input signals described in Section III-A2. τ_i and b_i are the time constants and biases for neuron i, respectively, w_{ij} is the weight matrix for interneuron connections, and w_{ij}^{IN} are the weights connecting the input elements to the neurons. The outputs (of which there are two in the applications considered here, representing the source voltages of the DC motors) are taken as the signals x_1 and x_2 obtained from the first two neurons. σ is the neuron activation function, here given by

$$\sigma(z) = \tanh(cz) \tag{9}$$

where c is a constant. Thus, the neuron signals are limited to the range $[-1,1]$, and the output must thus be rescaled to the appropriate range, as discussed in Section III-D.

b) Signal preprocessing: The final component associated with the RNNs used here is a signal preprocessing system (SPS), as shown in the left-hand side of the middle panel of Fig. 5. Biological organisms continuously process vast amounts of information, and the massive flow of input signals generally undergoes preprocessing (to reduce the volume of the flow) before reaching, e.g., the part of the brain responsible for balancing. Thus, augmenting the RNN model with a signal preprocessing unit is motivated by biological considerations.

The SPS is given by the simple equations

$$y_i = \sigma\left(\sum_{j=1}^{k} w_{ij}^{\mathrm{SPS}} z_j\right), \quad i = 1, \ldots, m \tag{10}$$

where z_j are the *raw input signals* and w_{ij}^{SPS} is the weight matrix connecting those signals to the *processed inputs signals* y_i (i.e., the input elements defined above). The entire system is shown in Fig. 5.

Note that, in all runs presented below, a strongly simplified SPS is used, in which there is a direct mapping between a raw input signal and the corresponding processed signal, according to

$$y_i = \sigma\left(w_{ii}^{\mathrm{SPS}} z_i\right), \quad i = 1, \ldots, m. \tag{11}$$

In this case, $w_{ij}^{\mathrm{SPS}} = 0$ if $i \neq j$. However, even such a simple SPS serves the important purpose of automatically (through the optimization of the w^{SPS} weights) scaling the input signals to appropriate ranges.

The SPS defined by (11) will be referred to as a *reduced SPS* in contrast to the *complete SPS* defined in (10).

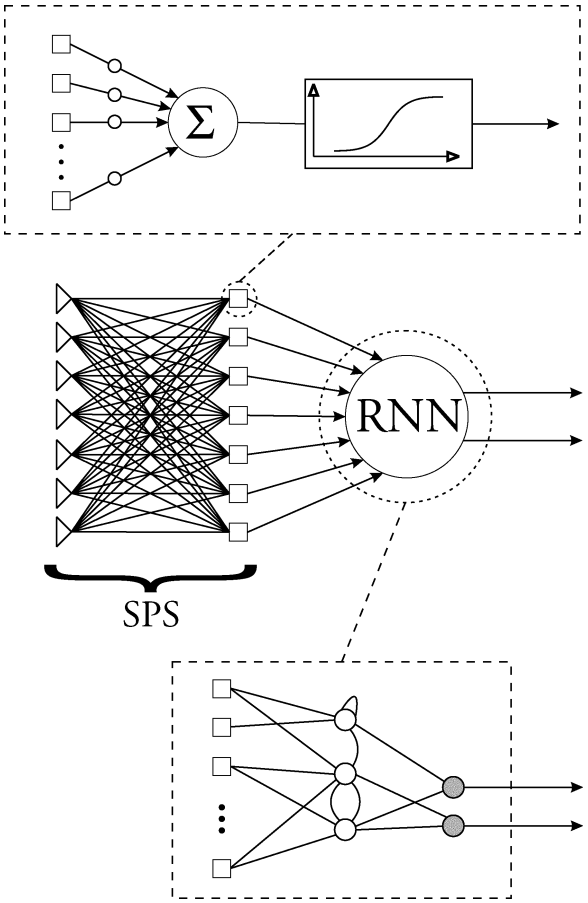


Fig. 5. RNN augmented with a signal preprocessing unit (SPS). In all figures showing neural networks, squares represent input elements providing processed input signals, triangles represent input elements providing raw input signals, filled circles represent output neurons, and nonfilled circles represent internal (i.e., nonoutput) neurons.

C. Parametric and Structural Optimization

In general, training an RNN is much more complex than training a simple feedforward neural network (FFNN) using, e.g., backpropagation. However, the use of an RNN rather than an FFNN is motivated in part by the fact that the balancing behavior must be equipped with a short-term memory to cope with cases in which two identical (or very similar) sets of instantaneous input signals (e.g., pressure distributions under the foot plate) require different responses. In an RNN, the motion just prior to the present state can be encoded, albeit implicitly, in the activation levels of its neurons.

Furthermore, the optimal structure of a neural network for balancing (or indeed for many other robotic behaviors) cannot easily be specified in advance. Most neural network methods are limited to parametric optimization, i.e., tuning of the weights in a pre-defined network architecture. However, if the initial specification of an RNN does not contain a sufficient amount of neurons for the problem at hand, it is possible that the solution will not be found at all. On the other hand, if instead the RNN is too large, the training of the network may become prohibitively time-consuming. Thus, ideally, the training algorithm should be capable not only of parametric optimization but structural optimization as well.

D. Evolutionary Optimization of RNNs

For a general introduction to EAs, see, e.g., [17]. In the standard genetic algorithm (GA) [18], which is one example of an EA, the variables of the problem are encoded in a fixed-length string. By contrast, the EA used here acts directly on the RNNs.

Basically, the structure optimized by the EA can be divided into two parts: 1) the weights w^{SPS} of the signal preprocessing system and 2) the neurons and their time constants, biases, and connections (w and w^{IN}).

Real-number encoding is used, i.e., all genes take floating-point values in the open interval [0,1], which are then rescaled to the appropriate range.

After each evaluation, a fitness value is assigned to each individual. The details of the evaluation procedure are described in Section IV-A. In the EA, generational replacement is used, i.e., all individuals are replaced by their offspring in each generation. The selection of individuals (for reproduction) is done by means of the tournament selection method, using a tournament size specified (at the start of a simulation) as a fraction of the population size. In addition, elitism is used, i.e., a single copy of the best individual is transferred without modification from one generation to the next.

In a standard GA, the selection step is followed by crossover (with a certain crossover probability) and mutation. In the EA used in this paper, a rich variety of operators for mutation and crossover can be defined, since the EA is designed to optimize both the parameters and the structure of the RNNs. Obviously, the selection of mutation and crossover operators, as well as the values of the associated parameters, will affect the performance of the EA. Thus, in Section V-A1, the specific choices made in this paper will be motivated briefly. Mutation operators can be divided into two categories: those that modify values of parameters, and those that modify the structure (i.e., the number of neurons or the number of connection weights) of the RNNs.

c) Parametric mutations: Here, two kinds of parametric mutations are defined, namely, *full-range mutations* that change the value of a weight to a new random value in the allowed range $[w_{\min}, w_{\max}]$, and creep mutations, in which the parameter is modified more gently according to

$$w^{\mathrm{new}} = w^{\mathrm{old}}(1 - 2rc + c) \tag{12}$$

where c is the creep rate and r is a random number in [0,1]. Note that the creep mutations can generate values (slightly) outside the interval allowed for full-range mutations. The *creep probability* (p_{creep}) determines the probability of using a creep mutation rather than a full-range mutation.

d) Structural mutations: Structural mutations of two kinds are defined here. *Connectivity mutations* either add or remove an incoming weight (either from a neuron or an input element) to a given neuron. In case of addition, the connecting neuron or input element is chosen randomly, and the weight is assigned a random value in the allowed range. *Neuron mutations*, on the other hand, add or remove entire neurons from the RNN. In case of removal, all outgoing and incoming links associated with the (randomly selected) removed neuron are also removed.

The case of addition is more complicated. In a small or moderately sized network, the addition of a neuron, with both

TABLE II
FITNESS MEASURES USED WHEN EVOLVING THE FOUR CONSTITUENT BEHAVIORS. SEE SECTION IV-A1 FOR THE DEFINITION OF THE y-DIRECTION

Behavior	Name	Fitness measure used in evolution
B1	Move forward	Distance moved in the positive y-direction.
B2	Move backward	Distance moved in the negative y-direction.
B3	Stop	Inverse of the integrated deviation from the upright position
B4	Charge battery	Inverse of the integrated deviation from the upright position

incoming and outcoming links, amounts to a macromutation, i.e., a mutation causing a significant alteration of the phenotype (the robotic brain, in this case). It is a well-known fact in evolutionary biology that macromutations generally have a detrimental effect, and the same holds in artificial evolution.

Thus, the addition of a neuron must be performed with care and, to that end, two different addition operators have been defined, namely: 1) *zero-weight addition*, which simply places an unconnected neuron (with random time constant and bias) in the network and 2) *single connection addition* in which a neuron, with random time constant and bias, is added with a single incoming connection from a randomly selected neuron or input element, and a single outgoing connection to a randomly selected neuron.

e) Crossover: Clearly, in view of the distributed nature of the computation in a neural network, it is far from trivial to define a useful crossover operator for such networks. Simply cutting a network into two parts, as one would do in the case of a simple GA chromosome, is unlikely to produce good results. In fact, such a procedure can be viewed as a huge macromutation. The situation is made even more complicated by the fact that different networks are, in general, of different size.

Several attempts were made to introduce a gentle crossover operator which would be able to combine material from two individuals without reducing the fitness of the resulting individual. However, as none of these crossover operators turned out to have a positive effect on the EA, they were not used further, and will therefore not be described here.

1) Simulation Package: All simulations were made using a simulation library called EVODYN, developed by one of the authors [19]. The rigid-body dynamics engine in EVODYN is based on Featherstone's algorithm [20], [21] and solves the equations of motion in time proportional to the number of links in the system. Developed in Delphi (Object Pascal), it supports tree-structured rigid-body systems described in terms of MDH parameters [22] and uses a fourth order Runge–Kutta method for numerical integration of the state derivatives of the simulated system.

The simulation software must also be able to maintain a population of structures of varying size, and also be able to perform the various operations on those structures that are needed by the evolutionary process. Such features are an integral part of EVODYN, simplifying the implementation of evolutionary optimization of robotic brains for motor behaviors.

The rigid body dynamics engine implemented in EVODYN runs significantly faster than real-time, making it useful in evolutionary applications where many evaluations are needed.

IV. EXPERIMENTAL PROCEDURE

In this section, the steps involved in generating the robotic brain will be described briefly, followed (in the next section) by a description of the actual results of the investigation. There are two main steps, namely: 1) the generation of individual behaviors and 2) the generation of the behavioral organizer.

A. Evolution of Behaviors

Here, behaviors were generated by means of an EA, and were represented as RNNs. All behaviors described here were evolved separately and independently of each other. Furthermore, during the evolution of individual behaviors, no obstacle or energy sensors were used, and the robots moved in an obstacle-free environment, as the problem of obstacle avoidance is one to be solved by the behavioral organizer (see next).

The four generated behaviors were *move forward* (B1), *move backward* (B2), *stop* (B3), and *charge batteries* (B4). The fitness measures used when evolving each of the four behaviors are summarized in Table II and the choice of input signals, provided to the RNNs representing each behavior, is described in Section V-A1

1) Behaviors for Forward (B1) and Backward (B2) Motion: Here, the task is to move the robot as far as possible during a limited amount of time. During evolution, these behaviors are awarded fitness according to the distance moved along the y-direction. The y-direction is the vertical axis going from the bottom of Fig. 1, pointing toward the top of the figure.

Both behaviors must be able to move the robot in the correct direction. If activated, B1 (and B2) moves the robot in a straight line, relative to the robots's starting position, i.e., there is no correction of heading involved. Also, both B1 and B2 must be able to start from a standstill, as the robot might have been brought to a stop by some of the other available behaviors.

The maximum simulation time was set to 4 s and the fitness measure was simply the distance traveled during that time. In the case of B1, the direction of motion (yielding positive fitness) was along the positive y-direction. The direction of motion in B2 was along the negative y-direction.

2) Stop Behavior (B3): The task here is to bring the robot to a halt. B3 should provide the robot with the ability to stop from either a moving or a stationary state. In the latter case, the task of B3 is to place the single leg of the robot in an upright position. Thus, B3 must be very robust in order to handle a large variety of initial starting positions and velocities. As long as B3 is activate, the robot will position itself in an upright, standstill position.

3) Battery Charging Behavior (B4): The charging behavior (B4) is similar to B3, the main difference being that it provides

the robot with the ability to charge its batteries. B4 need not be as robust as B3 provided that the robot has already reached a standstill position when B4 is activated. Making sure that this is the case, makes the task for the behavioral organizer even more challenging. The difference in the procedures used for generating B3 and B4, respectively, is described in greater detail in Section V-B.

B. Evolution of Behavioral Organization

Once the four constituent behaviors have been generated, the behavioral organizer can be generated using the UF method. In this paper, the task of the behavioral organizer will be to select between four behaviors (B1, B2, B3, and B4) to make the robot move through an arena, shown in Fig. 1, populated by moving obstacles. The arena is 8 m long and 3 m wide and the robot's starting position is located just outside one of shorter sides. Obstacles move with constant velocity across the arena, using periodic boundary conditions. That is, if an obstacle leaves the arena on one side, it reappears on the other. Initially, all obstacles are given directions of motion and initial positions using a uniform random number generator with a specific seed. Thus, all simulated robots are exposed to *exactly* the same environment, in order for the EA to progress smoothly. The risk of adaptation to the exact conditions encountered in this deterministic arena is lowered by the fact that the arena is quite long. However, some arena-specific adaptation did occur, as discussed below.

For the UF method, all utility functions (U_{1-4}) were given the following functional form:

$$U_i(\mathbf{v}) = a^{(i)} + \sum_{p=1}^{n} b_p^{(i)} \mathrm{v}_p + \sum_{p=1}^{n} \sum_{q=p}^{n} c_{pq}^{(i)} \mathrm{v}_p \mathrm{v}_q \qquad (13)$$

where $\mathbf{v}$ is a vector (with n elements) containing the values of external/internal sensors and internal abstract variables. $a^{(i)}$, $\mathbf{b}^{(i)}$, and $\mathbf{c}^{(i)}$ are constant parameters to be determined by the EA. Note that (for ease of notation), $a^{(i)}$ is a scalar, $\mathbf{b}^{(i)}$ is a vector, and $\mathbf{c}^{(i)}$ is an upper triangular matrix, representing a constant term, linear terms, and quadratic terms, respectively. Even though (13) can be of any form, experience with the UF method has shown that it is often sufficient to use a sum including constant, linear, and quadratic terms.

In the setup used here, $\mathbf{v}$ contains the values from three simple proximity sensors (s_1, s_2, and s_3), one battery sensor (E), and one internal abstract variable (x_i) for each behavior $i = 1, \ldots 4$. The internal abstract variables are (here) implemented such that $x_i \equiv 0$ *unless* behavior B_i is active.

The variation of the x_i is given by

$$x_i = k_{i1} + k_{i2} e^{-k_{i3} t_i} \qquad (14)$$

where k_{i1}, k_{i2}, and k_{i3} are constants and t_i is the behavior time (see Section III-B1), measuring the time since the last activation of behavior i. Both the fact that x_i is only nonzero in behavior i and the specific variation given by (14) represent rather arbitrary choices, motivated by experience with the UF method.

The x_i could have been specified differently, but the present specification introduces few additional parameters, and turns out to be sufficient for the problems considered here. The rationale for the introduction of the x_i is to give the EA *something* it can use, e.g., for preventing rapid behavior switching (see Section VI-D).

With the specific functional form chosen for the utility functions [see (13)] and the choice of variation for the internal abstract variables, the number of parameters used in each behavior can be calculated as

$$N = 1 + n + \frac{n^2 + n}{2} + n_x \qquad (15)$$

where n is the number of arguments in the utility functions [see (13)] and n_x is the number of parameters used in the expression for the internal abstract variable [see (14)]. Here, four different behaviors are used, each with a utility function that takes five arguments $(U_i(s_1, s_2, s_3, E, x_i))$, adding up to a total of 96 parameters to be determined by the EA.

In the EA, all parameters are randomly initialized in the range $[-1,1]$, to which they are constrained during the evolution. If a parameter receives a value outside this range, for instance via a creep mutation, it is forced to lie on the boundary of the range $[-1,1]$.

V. Results

In this section, the results of the investigation are presented, starting with an investigation of optimal parameter choices. Next, the evolution of the four individual behaviors is presented. Finally, the evolution of the behavioral organizer is presented, both for the simplified dynamical system and for the dynamically modeled robot.

A. Parameter Settings and Input Signal Selection

1) Optimal Parameter Choices: Before evolving behaviors or behavioral organization, a large set of runs was performed to find the best possible set of mutation operators and parameter choices. In general, finding optimal parameter values in problems of the kind considered in this paper is very difficult, since 1) the runs are rather time-consuming and 2) the results can vary quite strongly from run to run. Thus, in order to find a reliable average performance for a given setup, many runs must be performed. In order to make a fair comparison between different runs, the results after 100 generations were used. In all runs, the population size was equal to 50.

Some general conclusions could be drawn from these runs. First of all, the results turned out to be quite insensitive to the amount of creep mutations used, and to the relative creep length c. Second, the best performance was found in runs using a parametric mutation rate inversely proportional to the number of parameters in the network, and the best results were found for p_{mut} values in the range $[3/N_{\mathrm{parms}}, 5/N_{\mathrm{parms}}]$, where N_{parms} denotes the number of parameters that can be mutated.

The optimal rate of structural mutations $(p_{\mathrm{struct.mut}})$ was found to be around 0.1–0.3 p_{mut}. For the addition of neurons, the best results were found if zero-weight addition (see Section III-D) was performed with a probability of around 0.5–0.8 $p_{\mathrm{struct.mut}}$.

Based on these results, the following parameter set was chosen, and was then used throughout the simulations: $p_{\mathrm{mut}} = 3.5 \pm 0.5/N_{\mathrm{parms}}$, $p_{\mathrm{creep}} = 0.5$, relative

creep length $= 0.03$, $p_{\text{struct.mut}} = 0.3\, p_{\text{mut}}$, $p_{\text{zeroweight}} = 0.8$ (for neuron addition). The specification of p_{mut} indicates that this mutation rate was set either to $3/N_{\text{parms}}$ or $4/N_{\text{parms}}$. The optimal values found are those that would be expected: a mutation rate of $3/N_{\text{parms}}$ or $4/N_{\text{parms}}$ will, on average, lead to a few mutations per individual, a common optimum found when setting mutations rates [23]. The lower value found for the structural mutations is also natural, since those mutations generally make larger modifications of the evolving RNNs than the parametric mutations.

2) Input Signal Selection: Before evolving behaviors, a set of input signals presented to the RNN must be chosen. Using the parameter choices mentioned in the previous section, different sets of input signals were investigated. Possible input signals are contact forces under the foot $(\mathbf{F})$, joint positions $(\mathbf{X})$, joint velocities $(\mathbf{V})$, and acceleration of the COM $(\mathbf{A})$. Several runs were made with five different combinations of these possible signals: 1) $\{\mathbf{F}\}$; 2) $\{\mathbf{X}, \mathbf{V}\}$; 3) $\{\mathbf{F}, \mathbf{A}\}$; 4) $\{\mathbf{F}, \mathbf{X}, \mathbf{V}\}$; and 5) $\{\mathbf{F}, \mathbf{A}, \mathbf{X}, \mathbf{V}\}$.

As above, the EA used a population of 50 individuals, and was allowed to run for a total of 100 generations, allowing a fair comparison between different runs.

The results from these runs showed that the best set of input signals did indeed vary for different types of behaviors. When evolving a standstill (balancing) type behavior, the signal set that produced the best result was $\{\mathbf{X}, \mathbf{V}\}$. When a locomotion behavior was being evolved, the best results were obtained with the set $\{\mathbf{F}, \mathbf{X}, \mathbf{V}\}$, even though the set $\{\mathbf{X}, \mathbf{V}\}$ provided slightly better *average* results.

B. Evolution of Behaviors

As mentioned earlier, each behavior was evolved separately and independently. The parameters were set according to the results presented in Section V-A1. Guided by the results from the investigation of optimal input signals in the same section, two different sets were chosen for the *straight-line motion* behaviors (B1 and B2) and the *standstill* behaviors (B3 and B4). For the evolution of behaviors B1 and B2, the input signal set $\{\mathbf{F}, \mathbf{X}, \mathbf{V}\}$ was chosen, and in the case of B3 and B4, the set was $\{\mathbf{X}, \mathbf{V}\}$. In all runs, the EA used a population of 50 individuals and the architecture of each individual used a reduced SPS (see Section III-B2). In all runs, the initial networks contained between 3 and 12 neurons (random initialization), and were allowed to change in size as a result of the evolutionary process, although the minimum number of neurons was equal to two, so that motor signals always could be generated.

For all behaviors, the output of the first two neurons was taken as actuator signals and fed to the motors.

Several runs were made for each behavior. In this section, detailed descriptions are given of the networks that were chosen to be parts in the complete robotic brain.

1) Move Forward Behavior (B1): A typical run for the evolution of B1 lasted around 1000 generations. The final networks for B1 contained 10 ± 2 neurons (average over five runs). Once evolved, B1 was able to move the robot forward for the entire length of the simulation (4 s). In extended runs, B1 was able to keep the robot in a straight line for an additional 4 s, after which it started to deviate slightly from its forward path. However, the robot did not fall over. The demonstrated ability of B1 to keep the robot moving throughout runs that were much longer than those used during evolution, shows that the evolved B1 has solved the problem of forward motion in a general sense, rather than just adapting to the conditions during the first 4 s of the run. The cyclical nature of the torque curves (after the initial transient), see Fig. 6, also indicates that the evolved RNN displays an oscillatory behavior similar to that of a central pattern generator [24]. The motion of the robot (while using B1) had a jumping characteristic, where the robot used the bottom front edge of the foot to lean itself forward before pushing off and jump a short distance forward. The average power consumption for B1 was 12 J/s and the average velocity was 0.15 m/s. The final RNN, representing B1, consisted of eight neurons and is shown in Fig. 7. As is evident from the figure, the network was quite complex, and networks of similar complexity (not shown) were obtained for the other behaviors as well.

2) Move Backward Behavior (B2): Behavior B2 was evolved in the same way as B1 but displayed a significantly different type of motion. Here, the robot performed a crawling type of motion, by interchangeably using the corner points of the foot to move itself forward.

In the final network obtained for B2, the motion during B2 was straight-lined for the first 6 s, after which it started to turn. Thereafter, the robot kept turning, and after 12 s it had completed a full turn. However, during the time for which B2 was evolved (4 s), the robot displayed perfect straight-lined motion.

The slightly less impressive generalization properties of B2 were acceptable since, in the full robotic brain described next, B2 should normally only be used in emergencies, i.e., to move the robot backward in cases where forward motion is impossible and is, thus, not likely to be applied continuously for extended periods of time.

In B2, the average power consumption of the robot was 8.6 J/s and the average velocity during the straight-lined motion was 0.12 m/s. The final network representing B2 consisted of trn neurons.

3) Stop Behavior (B3): In B3, the task was simply to keep the robot upright. However, in the complete robotic brain described below, one can expect that the activation of B3 will normally occur when the robot is in motion, i.e., when it is executing either B1 or B2. Thus, the initial conditions for the activation of B3 may vary strongly from case to case, and the behavior must, thus, be sufficiently robust to handle many different initial conditions. In order to achieve such a behavior, the robot was subjected to perturbations (see below) throughout the evolution of B3.

a) Perturbations: The simulated robotic leg was subjected to perturbations in the form of nonperiodic torques added to the joints according to

$$\tau_1 = A\sin(kt) + B\cos(\sqrt{5}kt) \tag{16}$$

$$\tau_2 = C\cos(\sqrt{3}kt) + D\sin(\sqrt{2}kt) \tag{17}$$

if $(t \bmod 5) < 2$, and 0, otherwise, where t is the time elapsed since the start of the simulation, and k is a constant. The parameters A, B, C, and D were initialized to a value close to 0 at

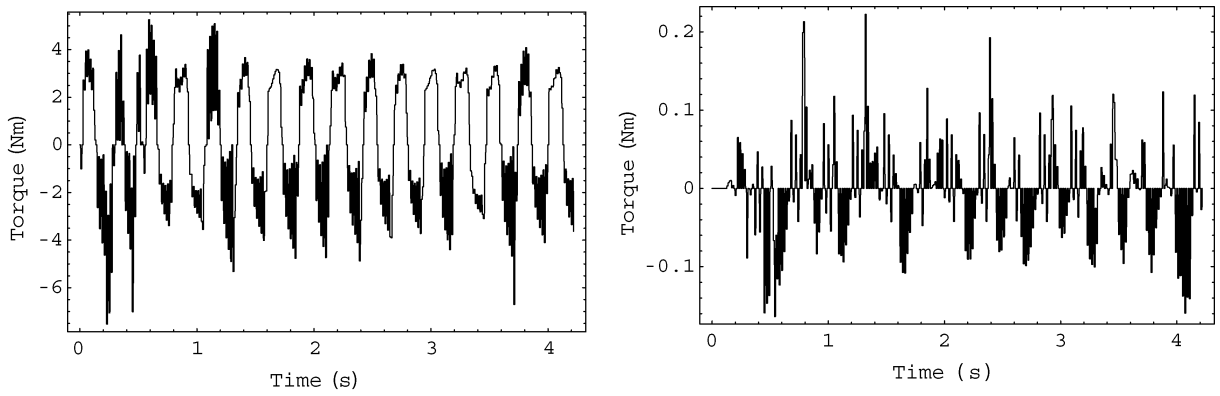


Fig. 6. The two torques, generated by the *move forward* behavior during the first 4 s. (a) Shows the torque applied to the lower joint of the robot's ankle. (b) Shows the torque applied to the upper joint.

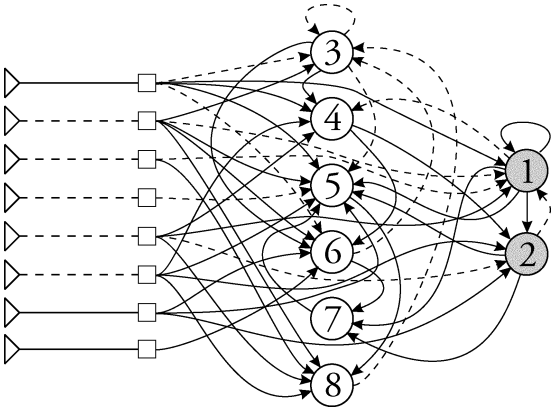


Fig. 7. RNN representing the *move forward* behavior (B1). Squares represent input units of the RNN and circles represent the neurons. Filled circles indicate output neurons (of which there are two here). Excitatory connections are shown as solid lines, and inhibitory connections (i.e., connections with negative weights) are shown as dashed lines. Triangles represent input units to the SPS.

the start of the simulation, and were then allowed to grow exponentially to a maximum value of 1 Nm. The exponential growth in these parameters took place in a stepwise manner after each period of nonzero perturbations.

The evolved behavior managed to keep the leg upright for the entire simulation length of 8 s. B3 proved to be very robust during extended simulations (also with perturbations, of the form described above), in which the robot was kept upright (and at standstill) for a total time of 120 s.

The average power consumption in B3 was 0.9 J/s and, at all times, the robot remained in the same position. The final network representing B3 consisted of eight neurons.

4) Charge Behavior: B4 was evolved in a similar way as B3, but without any external perturbations. Due to the absence of these perturbations, B4 was not able to perform as well as B3 in the extended simulations. However, for the 8 s used in the evolution of B4, the robot remained in an upright position. In extended simulations, the leg of the robot slowly moved out of position and toward the ground.

The final structure of the RNN representing B4 had 11 neurons. As in B3, the average power consumption in B4 was 0.9 J/s. Of course, the evolved RNN used for B3 could in principle have been employed also for B4. However, the fact that a less robust RNN was used for B4 makes the behavioral organization task more challenging.

C. Evolution of the Behavioral Organizer

Once the constituent behaviors had been evolved, several runs were made using B1, B2, B3, and B4 as the repertoire of behaviors made available to the behavioral organizer.

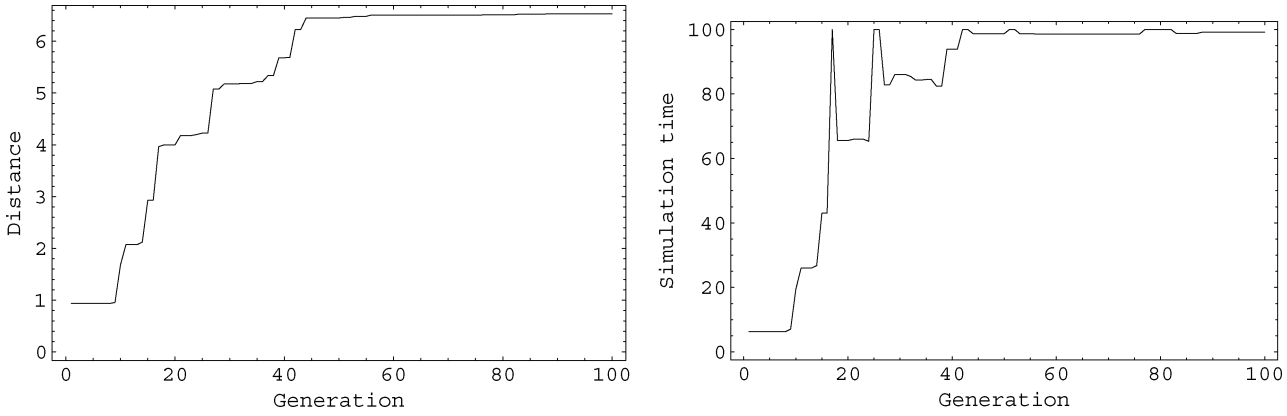
1) Simplified Model: First, and for verification of the UF behavioral organization method, several runs were made using the simplified model (see Section III-A3). The task of the robot was to move as far as possible across the arena, shown in Fig. 1, while avoiding collisions with obstacles and keeping the battery level from dropping to zero. In case of collisions or if the battery level reached zero, the simulation was terminated.

The size of the arena, (8 m $\times$ 3 m), was deliberately chosen such that it would generally be very difficult for the robot to traverse it entirely, since the aim was to test the ability of the robot to organize its behaviors while in a crowded environment, and while dealing with other problems, such as lack of battery energy.

Each behavior was associated with a utility function as described in Section III-B1 and Section IV-B, and the EA was, therefore, required to set the parameters of these functions, as well as the parameters determining the variation of the internal abstract variables (96 parameters in total), in order to solve the problem of crossing the arena. The fitness measure was simply taken as the distance moved in the positive y-direction.

In these runs, the evolved RNNs were not used to implement the behaviors B1, B2, B3, or B4. Instead, the behaviors were represented by constant velocities, which were set to match the average velocities of each evolved behavior (see Section V-B). For B1 the velocity was set to 0.15 m/s in the positive y-direction and for B2 the velocity was set to 0.12 m/s in the negative y-direction. The velocities for both B3 and B4 were set to zero. Therefore, in the simplified model, B3 and B4 differ only in the fact that, in B4, the robot gains energy, whereas in B3 it does not. Thus, a rational behavioral organizer would avoid using B3.

Since no dynamical (mechanical) system was implemented in this simplified system, each behavior had to be implemented with an artificial power consumption. As with the velocities, the



(a) (b)

Fig. 8. (a) Shows the distance traveled, as a function of generation (in the EA), for the best individual in a run with the simplified model. The maximum simulation time was 100 s. (b) The actual simulation time of the best individual is shown, again as a function of generation. This curve shows that, after around 40 generation, the best individual was capable of moving for the entire duration of the simulation without colliding or running out of energy.

power consumptions were set according to their RNN equivalents. That is, for B1, B2, B3, and B4, the power consumption was set to 12, 8.6, 0.9, and 0.9 J/s, respectively.

Due to the relatively small amount of available energy in the batteries, the behavioral organizer needs to activate the *charge* behavior quite often. The storage capacity of the battery and its properties during charging were deliberately chosen this way in order to test thoroughly the behavioral organization method.

In all runs performed, the evolved behavioral organizer was able to activate the different behaviors in such a way that the robot managed to avoid both collisions and running out of energy, for the entire length of the simulation. In early runs, the simulation time was set to 60 s. Once these runs were successfully completed, the simulation time was extended up to a maximum of 150 s. The storage capacity of the battery was normally 50 J, but in some runs it was raised to 75 or 100 J.

A representative case is illustrated in Fig. 8. In this simulation, the maximum simulation time was set to 100 s, and the best individual managed to move 6.5 m, giving it an average velocity of around 0.065 m/s. Due to the battery parameter settings chosen, i.e., the rather fast discharging of the battery, the robot did spend a lot of time in B4, which explains the rather low average velocity.

In Fig. 9, a snapshot of a typical situation is shown, in order to illustrate the operation of the evolved utility functions. Here, the robot was initially moving forward (top left panel), when it detected an obstacle in front at around $t = 96.5$ s. At this point, with the way forward being blocked, the behavioral organizer chose to drop the utility of B1 until the charging behavior became active ($t = 97$ s). Thus, while waiting for the obstacle to pass, the robot charged the batteries. Two factors then contributed to the reactivation of B1: The utility of B4 decreased (slightly) due to the rise of battery energy, and the passage of the obstacle (which disappeared from sight at around $t = 97.5$ s) raised the utility of B1, which was reactivated at around $t = 98.5$ s (top right panel).

In early runs, a fast switching phenomenon was observed in successful individuals. By switching between B1 and B4 at a high frequency, the behavioral organizer was able to regulate the speed of the robot while, at the same time, maintaining a high battery level. Needless to say, such a switching phenomenon would be less useful for the fully dynamical model (see below), since the acceleration phase present in B1 would then prevent the robot from moving at all.

Nevertheless, in order to avoid this problem, and also to generate a more realistic simulation, the constant T_0 in (7) was introduced, and was set to 0.5 s. The introduction of T_0 successfully removed the fast-switching phenomenon.

Another observation was that the organizer managed to keep the robot from getting caught in "traffic jams." That is, if the robot found itself (based on sensor values) in a situation with two obstacles approaching, one from each side, and a third obstacle blocking the forward direction, the organizer was able to activate B2 in order to avoid collisions. These situations did not occur frequently but when they did, the organizer handled the situation perfectly.

Also, since the fitness decreased during the time that B2 was active, successful individuals minimized the use of B2. Similarly, as expected, it was observed that B3 was never used by the simplified model.

a) Overfitting and validation: As is often the case in simulations involving optimization, the problem of overfitting, i.e., adaptation to a special situation, must somehow be tackled. Here, this has generally been done by evolving the simulated systems in an arena sufficiently long for the robot to encounter many different situations. However, some runs were also made in which each individual was evaluated in several (N_A) arenas with different initial obstacle configurations.[3] Increasing the number of evaluations per individual is a standard procedure for reducing overfitting. In these cases, the total fitness of the individual was taken either: 1) as the average performance (distance traveled) over the N_A arenas or 2) as the *worst* performance.

The results of several runs with different N_A are shown in Table III. As can be seen in the table, the runs with $N_A = 10$ showed better validation performance than those with smaller N_A, as expected. Common to all runs, however, is the fact that the validation performance was worse than the performance

[3]A specification of an obstacle configuration consists of setting the initial positions and directions of movement of all obstacles.

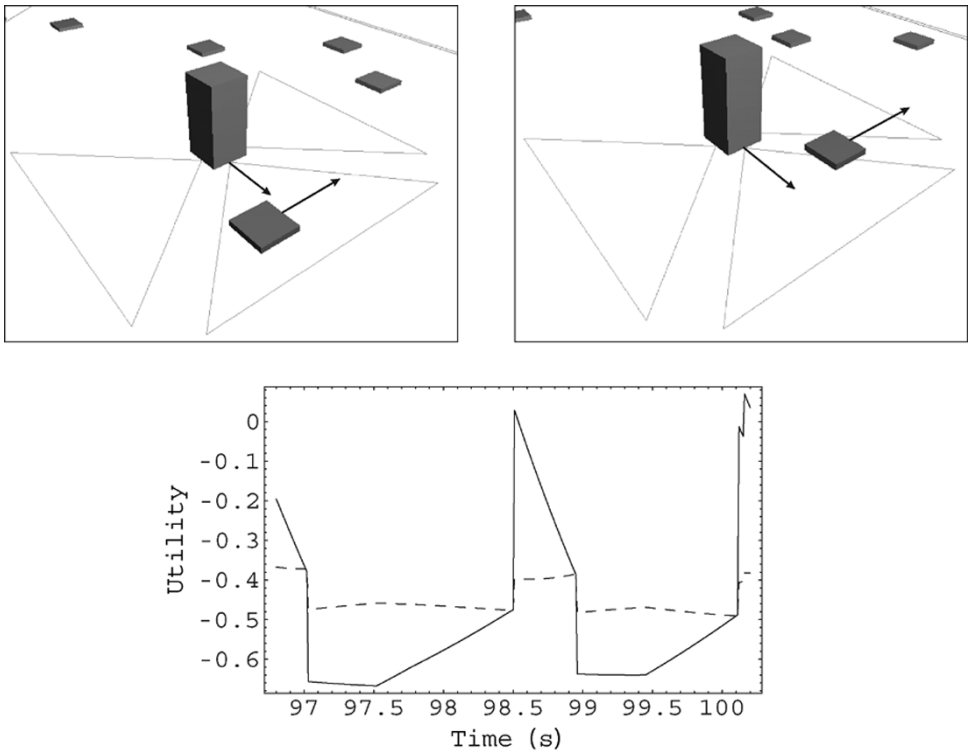


Fig. 9. A behavior switch sequence where, at first, behavior B1 (*move forward*) is active. Shortly thereafter, behavior B4 (*charge*) gets activated due to the detection of an approaching obstacle. As the obstacle passes, and the sensor signal decreases, B1 is activated again. For clarity, only the utility values for B1 (solid line) and B4 (dashed line) are shown in the bottom panel.

TABLE III
COMPARISON OF THE PERFORMANCE OF INDIVIDUALS EVOLVED AGAINST ONE, THREE, AND TEN DIFFERENT ARENAS, RESPECTIVELY. THE SECOND COLUMN SHOWS THE FITNESS TYPE (AVERAGE OR WORST) USED, THE THIRD COLUMN SHOWS THE AVERAGE FITNESS (i.e., DISTANCE MOVED) OBTAINED DURING TRAINING, THE FOURTH COLUMN SHOWS THE AVERAGE FITNESS (OVER 100 RANDOMLY GENERATED ARENAS) OBTAINED DURING VALIDATION, AND THE FIFTH COLUMN SHOWS THE CORRESPONDING STANDARD DEVIATION

N_A	Fitness type	$\overline{f}$ (tr.)	$\overline{f}$ (val.)	σ for $\overline{f}$ (val.)
1	—	10.23	3.87	2.32
3	average	11.16	3.96	2.10
3	worst	7.76	3.50	1.87
10	average	8.03	5.35	3.03
10	worst	9.95	5.82	3.22

during training, even though most individuals could traverse more than half of the arena regardless of the obstacle configuration encountered.

Runs where N_A is much larger than one are, of course, much more time-consuming than runs with $N_A = 1$ and, therefore, for the considerably slower runs involving the dynamical model (see next), only the single-arena case was studied.

2) Dynamical Model: Several runs were made using the full, dynamical model of the robot, again introducing utility functions for each behavior as described for the simplified model above, and again using the distance moved in the positive y-direction as the fitness measure.

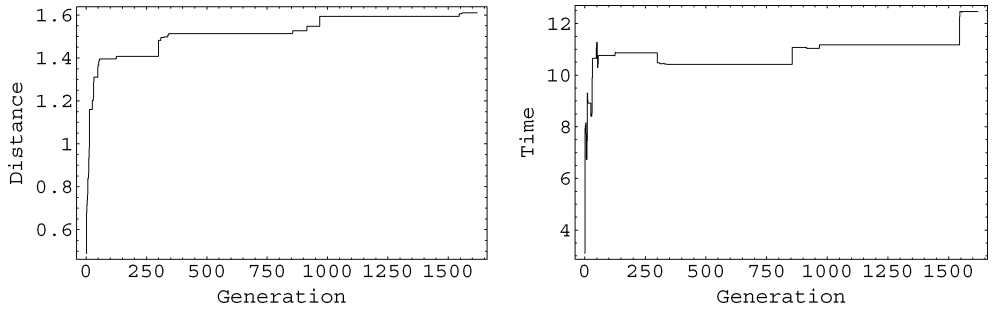
In these runs, the simulation time was set to 60 s, and the EA used a population size of 50 individuals. The parameters were set as above.

In the initial stages of all runs, it was common to find individuals that exclusively used B4 and, thus, were able to stand (in the same position) for the entire simulation. However, since they did not receive any fitness, those solutions were quickly removed by the EA. As the runs progressed, individuals appeared that were able to move and, eventually, also start passing the moving obstacles. However, in the case of the dynamical model, the performance of the simulated robots was much worse than for the simplified model.

Fig. 10 illustrates the performance of the best individual as a function of generation for one of the more successful runs. As can be seen in the figure, the best individual was able to move 1.6 m before it ran out of energy (the battery level was set to 100 J) after a total time of 12 s.

Even though the best individuals moved for a relatively short time, compared with the best individuals obtained with the simplified model, the evolved behavioral organizer made the robot efficiently use its sensors to avoid the moving obstacles. Fig. 11 shows a sequence of screenshots from the early stages of the motion of the best individual in a typical run, and the variation of the four utility functions (bottom right panel) representing B1, B2, B3, and B4.

Fig. 11 shows how the behavioral organizer uses the available sensors to avoid colliding with an obstacle. The top left panel shows the robot while it is moving forward (B1 is active) and, at the same time, an obstacle is approaching from its left side (i.e., the right side in the figure). As the robot moves forward, the energy level decreases, and near $t = 3$ s, a switch is made to B4 (charge) and the robot stops (a nontrivial procedure for the dynamically modeled robot). In the top right panel, the robot is



(a) (b)

Fig. 10. A typical run using the fully dynamical robot. (a) Shows the distance travelled by the best indiviual in each generation. (b) Shows the actual simulation time of the best individual is shown for each generation. As the maximum simulation time was set to 60 s, the solutions shown here were only able to complete a small part of that time (12 s). This is due to the fact that simulations were aborted in case the robot fell over, ran out of energy, or if a collision with an obstacle occurred. In this particular case, the robot ran out of energy.

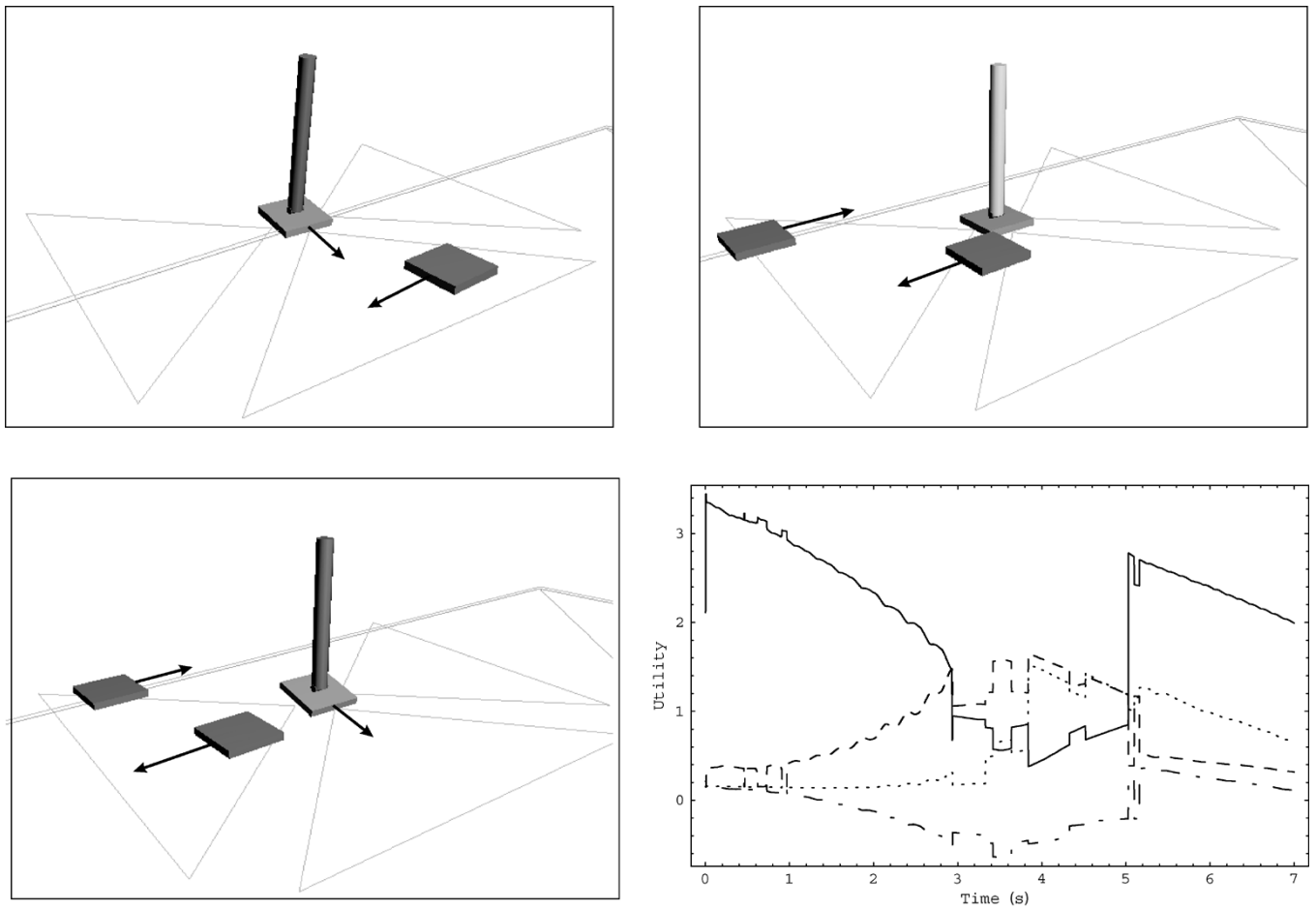


Fig. 11. Early stages of a typical run. The bottom right panel shows the variation of the four utility functions in one of the best performing individuals during evolution with the dynamic model. The curves represent the utility values for the behaviors *move forward* (solid curve), *move backward* (dotted curve), *stop* (dash-dotted curve), and *charge* (dashed curve). See Section V-C2 for a detailed discussion of this figure.

standing still, charging its batteries. After an additional 2 s (at $t = 5$ s), the front proximity sensor signal makes a sharp drop due to the fact that the obstacle moves out of the sensor's range and the organizer then activates B1, making the robot move forward again (bottom left panel).

It is important to note that the drop in utility for B2 at around 5 s is affected both by the increase in battery energy and the fact that the obstacle moves out of range of the sensor. Turning the sensors off leads to disastrous results (collision), demonstrating that the activation of behaviors is *not* a result of, e.g., lucky timing caused by adaptation to the situation at hand, but instead, as it should be, an active choice based on the available sensory information.

This demonstrates the strength of the behavioral organization method and shows that it is possible to combine independently evolved behaviors into a complete robotic brain. However, the fact that the robot had problems repeating the success in the runs made with the simplified model (see Section V-C1) points to-

ward the organizer's difficulties in coping with the imperfect behaviors available in the dynamical case (see Section VI) in conjunction with the properties of the robot (i.e., the low storage capacity and fast discharging of the battery). In fact, the evolution of the behavioral organizer was quite successful, as the robot managed to do the best it could, given the situation.

Animations and screenshots, produced during the work with this paper, can be downloaded from http://www.me.chalmers.se/~mwahde/robotics/UFMethod/Locomotion/hopping.html.

VI. Discussion

A. Behavioral Organization Using the UF Method

The main conclusion that can be drawn from the experiments performed here, is that the UF method is indeed able to organize successfully a set of behaviors, starting from random utility functions. Even more importantly, the method is able to make good use of behaviors that have *not* be tailor-made for the particular application in which they are put to use. For example, none of the behaviors B1–B4 were evolved using information from proximity sensors. Instead, the ability of the final robotic brain to avoid obstacles is an emergent property of the behavioral organizer.

In addition, it can be concluded that, through the optimization of the utility functions, the UF method allows the robot to execute behaviors that do not generate a fitness increase. Indeed, in one case (B2), even a behavior that *decreases* the fitness is used when needed. Of course, failure to use the auxiliary behaviors B2–B4 would lead to lower fitness in the long run, since the robot would run out of energy or would collide with obstacles. However, the important point is that *the UF method solves the problem of continuous relevance assignment for different behaviors*, at least in cases where the robot has a single main task to perform. Expressed differently, in the UF method, it is *not* required that the user be able to specify, by hand, the relative importance of, say, charging batteries and moving forward.

B. Performance Limitations

As mentioned in Section V, the size of the arena was chosen such that it would be very difficult for the robot, with its given maximum simulation time, to traverse it completely. Nevertheless, the simplified robot, infact managed to traverse the arena in several cases, whereas the dynamically modeled robot performed worse. However, this decrease in performance can be attributed to the fact that the dynamics of this robot is, per definition, much more complex than that of the simplified robot. Thus, even if the behavioral organizer achieves near-perfect selection of behaviors, the robot's motion will be limited by the capabilities of the constituent behaviors. Thus, an important (though somewhat trivial) conclusion is that the capabilities of a robotic brain depend not only on the behavioral organizer, but also on the individual behaviors, and that even the best behavioral organizer can fail to achieve its goals if the constituent behaviors are inadequate.

Furthermore, some of the failures of the behavioral organizer in the case of the dynamical model were due to rare situations that were simply impossible for the simulated robot to handle *regardless* of the behavioral organizer used. An example is shown in Fig. 12. In the situation shown in the figure, the robot is initially moving forward when it is disturbed by an incoming obstacle. Passing the first obstacle, the robot continues to move forward, only to discover a previously unseen (due to limited sensor range) obstacle, forcing the robot to continue forward, at which point *another* obstacle (the top one in the figure) enters the sensor range, again preventing the robot from stopping. Despite the perfect performance of the behavioral organizer, the robot then failed, due to an empty battery. Thus, the failure was, in this case, entirely a result of the specific settings used for sensor range and battery level.

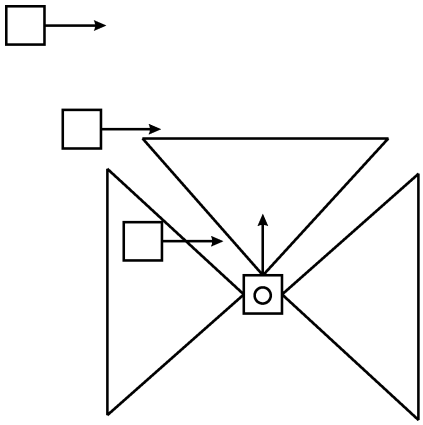


Fig. 12. Illustration of an obstacle configuration that is impossible to pass (due to the limited amount of energy available in the battery, with the settings used here), yet very difficult to avoid (due to the limited sensor range).

C. Performance During Validation Runs

The increase in validation performance as N_A was increased, also shown in Table III, indicates that the use of several different obstacle configurations is needed in order to find the best possible behavioral organizers, even though those obtained with $N_A = 1$ at least managed to pass around half of the arena regardless of the encountered obstacle configuration.

D. Behavioral Organization, Evolution, and Ethology

Evidently, the task of organizing several behaviors is by no means trivial, since the utility functions and internal abstract variables that determine the behavioral organization usually come with a large number of parameters (96, in the case considered here). The many parameters is one of the main motivations for using an EA for the optimization, and it does indeed (normally) arrive at a solution rather quickly, in terms of the number of evaluated individuals. For example, for the simplified models, successful behavioral organizers could be obtained after the evaluation of around 5000 individuals (see Fig. 8), which required around 2.5 hours on a P4 computer with a 2.53 GHz processor (in cases where each individual was evaluated in a single arena. The running time scales linearly with the number of evaluations per individual). The runs involving the dynamical model required more individuals: in some cases, up to 50 000 individuals were evaluated, in which case the runs took several days.

It should be noted that the EA is, to a great extent, parallelizable, meaning that a cluster of N computers could speed up a

run by (slightly less than) a factor N. However, it is doubtful whether more extensive runs would improve the results, since the performance of the evolved behavioral organizers was found to be limited not so much by the number of evaluated individuals, but rather by the quality of the constituent behaviors (see Section VI-A).

The UF method [10] is based on ethological considerations. Ethology is (or should be) an important source of inspiration for (autonomous) robotics, since animals often show remarkably efficient solutions to the problem of behavioral organization (see [25] for an excellent review). In particular, animals, (and, for that matter, humans) behave *as if* they were maximizing a quantity, which one may call *utility*, and which serves as the common currency when a decision is made concerning which behavior to activate in a given situation.

The behavioral organizers obtained via the UF method are designed always to activate the behavior associated with maximum utility. However, some behaviors, such as battery charging, usually display sinking utility values when activated, often leading to a rapid switching back and forth between, say, a charging behavior and a motion behavior. Taking a hint from ethology (see, e.g., [26, p. 72]), such switching should be avoided: animals normally do not switch rapidly back and forth between different behaviors, unless it is absolutely necessary to do so, since such switching normally reduces the overall performance. Thus, the ansatz for a utility function must be such that it will be *possible* for the behavioral organizer to avoid rapid switching. This is one of the motivations for the introduction of the abstract internal variables: the sudden jumps in their values when a new behavior is activated reduces the risk for dithering. Indeed, none of the evolved behavioral organizers obtained here displayed rapid behavior switching, except for the early runs with the simplified model. However, in that case, the behavior switching was not caused by badly designed utility functions and internal abstract variables, but rather by the unrealistically simple charging behavior. Once the delay T_0 was introduced, all dithering disappeared immediately.

E. Evolution of Individual Behaviors

In Section V, it was noted that the dynamically modeled robot achieved worse results than the simplified model. While this is not surprising, it points to a difficulty in using RNNs as an architecture for motor behaviors. This is a problem of robustness: even in careful evolution, where the evolving systems are exposed to many different situations, the RNNs rarely become sufficiently general to function in any situation (as mentioned in Section V-B). It is interesting to note, however, that the behavioral organizer was still able to do fairly well, due to its ability to keep behaviors active only for as long as they performed as intended. On the other hand, this represents an additional (and somewhat unnecessary) complication for the behavioral organizer: it would certainly be better if the behaviors were always performing their tasks well, so that the organizer could instead focus completely on the selection of behaviors based on more relevant factors (such as, e.g., battery energy, the presence or absence of obstacles, etc.).

VII. Conclusion

In this paper, it has been demonstrated how a complete robotic brain for single-legged locomotion can be generated by means of a two-stage process, in which a repertoire of behaviors are generated first, by whatever means desired (in this case, artificial evolution of RNNs), after which the behavioral organizer is generated using the UF method. The utility functions, obtained through artificial evolution in the UF method, allows the robotic brain to select correctly between the available behaviors in order to solve the primary task of the robot. It has also been demonstrated that the UF method is able to solve the behavioral organization problem even in cases where the constituent behaviors are not completely reliable in their performance over extended periods of time, as was generally the case for the RNN-based motor behaviors considered here. Finally, the importance of selecting an appropriate repertoire of behaviors, as well as equipping the robot with appropriate capabilities as regards, e.g., sensors and batteries has also been demonstrated. If the behavioral repertoire (or, e.g., the sensory capabilities) are chosen badly, the robot may fail in its task even if the behavioral organization is carried out perfectly under the given circumstances.

References

[1] T. Arakawa and T. Fukuda, "Natural motion trajectory generation of biped locomotion robot using genetic algorithm through energy optimization," in *Proc. IEEE Int. Conf. Systems, Man, Cybern.*, 1996, pp. 1495–1500.

[2] J. Furusho, S. Akihito, S. Masamichi, and K. Eichi, "Realization of bounce gait in a quadruped robot with articular-joint-type legs," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1995, pp. 697–702.

[3] K. Hirai, M. Hirose, Y. Haikawa, and T. Takenaka, "The development of Honda humanoid robot," in *Proc. IEEE Int. Conf. Robotics Automation*, 1998, pp. 1321–1326.

[4] C. Paul and J. Bongard, "The road less travelled: morphology in the optimization of biped robot locomotion," in *Proc. IEEE/RSJ Int. Conf. Intell.Robots Syst.*, Maui, HI, 2001, pp. 226–232.

[5] M. Wahde and J. Pettersson, "A brief review of bipedal robotics research," in *Proc. 8th UK Mechatronics Forum Int. Conf.*, Jun. 2002, pp. 480–488.

[6] Y. Fujimoto and A. Kawamura, "Simulation of an autonomous biped walking robot including environmental force interaction," *IEEE Robot. Autom. Mag.*, vol. 5, no. 2, pp. 33–42, Jun. 1998.

[7] Q. Li, A. Takanishi, and I. Kato, "Learning control of compensative trunk motion for biped walking robot based on ZMP," in *Proc. IEEE/RSJ Int.Conf. Intell. Robot Syst.*, vol. 1, 1992, pp. 597–603.

[8] B. Blumberg, "Action-selection in Hamsterdam: Lessons from ethology," in *Proc. 3rd Int. Conf. Simulat. Adapt. Behav.*, Brighton, U.K., Aug. 1994, pp. 108–117.

[9] P. Pirjanian, "Behavior-coordination mechanisms—State-of-the-art," Inst. Robotics and Intell. Syst., Univ. Southern California, Los Angeles, CA, Tech. Rep. IRIS-99-375, Oct. 1999.

[10] M. Wahde, "A method for behavioral organization for autonomous robots based on evolutionary optimization of utility functions," *J. Syst. Control Eng.*, vol. 217, no. 4, pp. 249–258, Sep. 2003.

[11] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[12] MicroMo. [Online]. Available: http://www.micromo.com

[13] Faulhaber. [Online]. Available: http://www.faulhaber.com

[14] J. Pettersson, H. Sandholt, and M. Wahde, "A flexible evolutionary method for the generation and implementation of behaviors for humanoid robots," in *Proc. IEEE-RAS Int. Conf. Humanoid Robotics*, Nov. 2001, pp. 279–286.

[15] L. Magdalena and F. Monasterio-Huelin, "A fuzzy logic controller with learning through the evolution of its knowledge base," *Int. J. Approx. Reasoning*, vol. 16, no. 3–4, pp. 335–358, Apr.–May 1997.

[16] C. Paul, "Bilateral decoupling in the neural control of biped locomotion," in *Proc. 2nd Int. Symp. Adapt. Motion Animals Mach.*, Kyoto, Japan, 2003.

[17] J. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press, 1992.
[18] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1996.
[19] J. Pettersson, "Evodyn: A simulation library for behavior-based robotics," Dept. Mach. Vehicle Syst., Chalmers Univ. Technol., Göteborg, Sweden, Tech. Rep., Sep. 2003.
[20] R. Featherstone, *Robot Dynamics Algorithms*. Norwell, MA: Kluwer, 1987.
[21] S. McMillan, "Computational dynamics for robotic systems on land and under water," Ph.D. dissertation, The Ohio State Univ., Columbus, OH, Summer 1994.
[22] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 2nd ed. Reading, MA: Addison-Wesley, 1989.
[23] G. Ochoa, I. Harvey, and H. Buxton, "Optimal mutation rates and selection pressure in genetic algorithms," in *Proc. Genetic Evol. Comput. Conf.*, 2000, pp. 315–322.
[24] T. Reil and P. Husbands, "Evolution of central pattern generators for bipedal walking in a real-time physics environment," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 159–168, Apr. 2002.
[25] D. McFarland, *Animal Behavior: Psychobiology, Ethology and Evolution*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, Dec. 1998.
[26] P. Slater, *Essentials of Animal Behavior*. Cambridge, U.K.: Cambridge Univ. Press, 1999.

Jimmy Pettersson received the B.Sc. degree in mechanical engineering from Mälardalens Högskola, Eskilstuna, Sweden, in 1997, and the M.Sc. degree in mechanical engineering from Chalmers University of Technology, Göteborg, Sweden, in 1999. He is currently working towards the Ph.D. degree in the Adaptive Systems Group, Department of Applied Mechanics, Chalmers University of Technology.

His main research concerns the use of biologically inspired computation methods in the field of behavior-based robotics.

Mattias Wahde received the Ph.D. degree in mechanics from Chalmers University of Technology, Göteborg, Sweden, in 1997.

He is currently an Associate Professor and leads the Adaptive Systems Group, Department of Applied Mechanics, Chalmers University of Technology. His main research interests are biologically inspired computation methods and their applications, particularly, in the fields of autonomous robots and bioinformatics.

Paper IV

Structural Evolution of Central Pattern Generators for Bipedal Walking in 3D Simulation

to appear in

Proceedings of the 2006 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2006), Taipei, Taiwan, October 2006.

Structural Evolution of Central Pattern Generators for Bipedal Walking in 3D Simulation

Krister Wolff, Jimmy Pettersson, Almir Heralić, and Mattias Wahde

Abstract— **Anthropomorphic walking for a simulated bipedal robot has been realized by means of artificial evolution of central pattern generator (CPG) networks. The approach has been investigated through full rigid-body dynamics simulations in 3D of a bipedal robot with 14 degrees of freedom. The half-center CPG model has been used as an oscillator unit, with interconnection paths between oscillators undergoing structural modifications using a genetic algorithm. In addition, the connection weights in a feedback network of predefined structure were evolved. Furthermore, a supporting structure was added to the robot in order to guide the evolutionary process towards natural, human-like gaits. Subsequently, this structure was removed, and the ability of the best evolved controller to generate a bipedal gait without the help of the supporting structure was verified. Stable, natural gait patterns were obtained, with a maximum walking speed of around 0.9 m/s.**

I. INTRODUCTION AND MOTIVATION

The great interest in humanoid robots during the last decade is motivated by the many advantages of bipedal robots over wheeled robots. First of all, humanoid robots (and bipedal robots in general) are able to move in areas that are inaccessible to wheeled robots, such as staircases and rugged outdoor terrain. In addition, their human-like shape allows such robots to function in constructed environments, such as homes or industries which, naturally, are adapted to people. Furthermore, recent studies [1], [2], [3] have claimed that people are more comfortable interacting with a robot with an approximately human shape, rather than a tin can-like wheeled robot.

However, an obvious problem confronting humanoid robotics is the generation of stable gaits. Whereas wheeled robots normally are statically balanced and remain upright regardless of the torques applied to the wheels, a humanoid robot must be actively balanced, particularly if it is to execute a human-like, dynamic gait. Several methods for generating bipedal gaits have been proposed in the literature. An important example is the ZMP method [4], where control torques are generated in order to keep the zero-moment point within the convex hull of the support area defined by the feet.

However, the success of gait generation methods based on classical control theory, such as the ZMP method, relies on the calculation of reference trajectories for the robot to follow. That is, trajectories of joint angles, joint torques, or the centre-of-mass of the robot are calculated so as to satisfy the ZMP constraint [5], [6]. When the robot is acting in a well-known constructed environment, the ZMP method should work well. When acting in a dynamically changing real world environment, however, the robot will encounter unexpected situations which cannot all be accounted for beforehand. Hence, reference trajectories can rarely be specified under such circumstances. To address this problem, there has recently been a movement in the robotics community towards alternative, biologically inspired control methods. Such methods do not, in general, require any reference trajectory. Typically, robotics researchers employ bio-inspired control strategies based on artificial neural networks (ANNs) [7], [8] or central pattern generators (CPGs) [9]. Often some kind of evolutionary algorithm (EA) is utilized for the design of the controller [10], [11], [12], [13], and [14].

Clearly, walking is a rhythmic phenomenon, and many biological organisms are indeed equipped with CPGs, i.e. neural circuits capable of producing oscillatory output given tonic (non-oscillating) activation [15]. CPGs have been studied in several simple animals, such as the lamprey [16] for which mathematical models have been developed as well [17], [18]. CPGs have also been studied in more complex animals, such as cats and primates ([19], [20], [21]), and there are also observations that support the notion of CPGs in humans. For example, treadmill training of patients with spinal cord lesions is assumed to rely on the adequate activation of a CPG [21].

Developing artificial counterparts to biological CPGs, with the aim of generating robust gaits for bipedal robots, is an active field of research. In seminal works by Taga *et al.*, [9], [22], a gait controller based on the half-center CPG model (see below) has been investigated. It was demonstrated in a 2D simulation of a five-link biped that the controller made the robot robust against physical perturbations [9]. Furthermore, obstacle avoidance through regulation of the step length was realized [22].

Shan *et al.* [11] generated bipedal walking in a 2D simulation using CPGs. A multi-objective genetic algorithm was used to optimize the synaptic weights in a network composed of nine CPG units. Reil and Husbands [23] used genetic algorithms (GAs) to optimize fully connected recurrent neural networks (RNNs), which were used as CPGs to generate bipedal walking in 3D simulation. They used a GA, with a real-valued encoding scheme, to optimize weights, time constants, and biases in fixed architecture RNNs. Their biped model had six degrees-of-freedom (DOFs), and consisted of a pair of articulated legs connected with a link. The resulting CPGs were capable of generating bipedal, straight-line walking on a planar surface. Furthermore, simple sensory input to

The authors are affiliated with the Department of Applied Mechanics, Chalmers University of Technology, 412 96 Göteborg, Sweden. Corresponding author's e-mail: krister.wolff@chalmers.se

locate a sound source was integrated to achieve directional walking.

CPGs have desirable properties, such as intrinsic aptness for the formation of periodic output patterns and adaptation to the environment through entrainment, for the generation of gaits and other types of repetitive and stereotypic motions.

Manually tuning the parameters of the CPGs and defining the feedback and interconnection paths in an optimal way is a daunting task. In many cases reported in the literature, e.g. [22], [24], [25], [26], and [27], the design of CPG networks has commonly been carried out in an intuitive manner; a time-consuming and difficult process which may lead to sub-optimal performance. Even in cases where GAs have been applied, as in several of the references mentioned above, the approach has generally been restricted to parametric optimization in a network of fixed architecture.

In this paper, the problem of generating both the structure, i.e. the network feedback and interconnection paths, and the parameters of a CPG network controlling a fully three-dimensional, simulated bipedal robot with 14 DOFs will be considered, using a GA as the optimization method. The half-center CPG model, as originally proposed by Matsuoka [28], will be adopted as the oscillator unit. A challenging problem, which is seldom mentioned (the papers by Paul and Bongard [29] are an exception), is the fact that, while biological organisms have developed their walking patterns (and, indeed, other behaviors as well), over long periods of simultaneous evolutionary optimization of both body and brain, in robotics one attempts instead to provide an already fixed body structure with a brain capable of generating a bipedal gait. This poses many problems for a GA-based approach. For example, if only the distance covered is used as the fitness measure, a common result is to find individuals that simply throw themselves forward, rather than walking; Walking would certainly yield a higher fitness value, yet this solution may be very hard to find, given the readily accessible local optimum found by those individuals throwing their body forward. Thus, the evolutionary process grinds to a halt almost immediately. Of course, this type of solution can be avoided simply by adding constraints on body posture as part of the fitness measure. However, such constraints must often be added in an *ad hoc* manner, and they often lead to results (such as non-natural gaits) that are undesirable. Rather than changing the fitness measure, one may attempt to change the body of the robot. Evolving an upright, bipedal gait from, say, an initial population of crawling individuals would perhaps be infeasible. However, another option, which will be considered in this paper, is to add a supporting structure to the robot, helping it to balance as it starts to walk. Some different strategies for subsequently removing this support, while maintaining a dynamically stable gait, will then be investigated.

II. CENTRAL PATTERN GENERATORS

A. Models from biology

From biological studies, three main types of neural circuits for generating rhythmic motor output have been proposed,

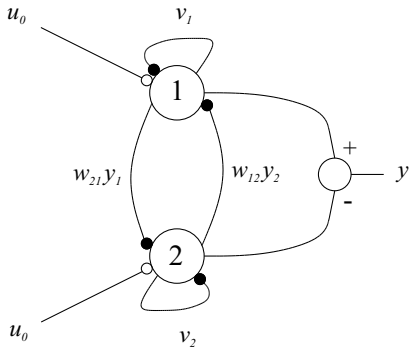


Fig. 1. A half-center model (Matsuoka) oscillator unit. Excitatory connections are indicated by open circles, and inhibitory connections are indicated by filled disks.

namely the *closed-loop model*, the *pacemaker model*, and the *half-center model*. The two former models are described in [30]. The half-center model, which will be considered in this paper, was proposed to account for the alternating activation of flexor and extensor muscles of the limbs of a cat during walking. The basis of this model is the classical experiments reported by Brown from 1911 [31] and 1912 [19]. Each pool of motor neurons for flexor or extensor muscles is activated by a corresponding half-center, or pool, of interneurons. Another set of neurons provides for a steady excitatory drive to these interneurons. Between each pool of interneurons are inhibitory connections which ensure that, when one pool is active, the other is suppressed. Matsuoka [28] analyzed the mutually inhibiting neurons and found the conditions under which the neurons generated oscillations.

B. Mathematical formulation of the CPG model

Commonly, a CPG is computationally modeled as a network of identical systems of differential equations, which are characterized by the presence of attractors[1] in the phase space [32]. Usually, a periodic gait of a legged robot is a limit cycle attractor, since the robot periodically returns to (almost) the same configuration in phase space.

Each node in the network is referred to as a neuron, or cell. The half-center model mentioned above is commonly adopted as the biological foundation for a rhythm generator, see e.g. [9], [11], [22], [24]. The neurons in the half-center model are described by the following equations [9]:

$$\tau_u \dot{u}_i = -u_i - \beta v_i + \sum_{j=1}^{n} w_{ij} y_j + u_0, \quad (1)$$

$$\tau_v \dot{v}_i = -v_i + y_i, \quad (2)$$

$$y_i = \max(0, u_i), \quad (3)$$

where u_i is the inner state of neuron i, v_i is an auxiliary variable measuring the degree of self-inhibition (modulated by the parameter β) of neuron i, τ_u and τ_v are time constants, u_0 is an external tonic (non-oscillating) input, w_{ij} are the weights connecting neuron j to neuron i, and, finally, y_i

[1] Attractors are bounded subsets of the phase space, to which regions of initial conditions converge as time evolves.

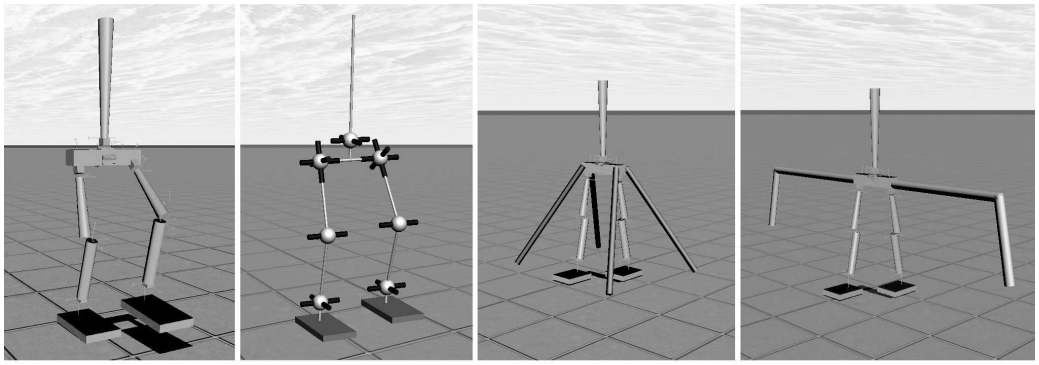


Fig. 2. The leftmost panel shows the simulated robot, and the second panel from the left shows its kinematics structure with 14 DOF. The two right panels show the robot with its supporting structure, the left one having four contact points, while the right one has two contact points.

is the output of neuron i. Two such neurons arranged in a network of mutual inhibition (a half-center model), as shown in Fig. 1, form an oscillator, in which the amplitude of the oscillation is proportional to the tonic input u_0. In addition, if an external oscillatory input is applied, the oscillator will lock to the frequency of the input. If the input is removed, the oscillator smoothly returns to the original frequency.

III. METHOD

In this section the physical simulation environment, the CPG network structure, the feedback paths, and the evolutionary algorithm will be described.

A. Dynamical simulation

A fully three-dimensional bipedal robot with 14 degrees of freedom, shown in the leftmost panel of Fig. 2, was used in the simulation experiments. The simulated robot weighs 7 kg and is 0.75 m tall. The distance between the ground and the hips is 0.45 m. As shown in the second panel from the left in Fig. 2, the waist has 2 DOFs, each hip joint has 3 DOFs, the knee joints have 1 DOF each, and the ankle joints have 2 DOFs each. The CPG network generates torques, which are applied to their respective joints in order to control the robot. To guide the evolution towards a natural biped gait, the robot has been fitted with two different mass-less posture-support structures, as depicted in the two right panels of Fig. 2.

The simulations were carried out using the EvoDyn simulation library [33], which was developed at Chalmers University of Technology. Implemented in object-oriented Pascal, EvoDyn is capable of simulating tree-structured rigid-body systems and runs on both Windows and Linux platforms. Its dynamics engine is based on a recursively formulated algorithm that scales linearly with the number of rigid bodies in the system [34]. For numerical integration of the state derivatives of the simulated system, a fourth order Runge-Kutta method is used. Visualization is achieved using the OpenGL library.

B. CPG network

In the CPG network, which is responsible for the generation of motions, each joint is assigned a specific half-center oscillator consisting of two neurons; a flexor neuron and an extensor neuron. The overall structure of the CPG network is depicted in Fig. 3.

In order to reduce the size of the search space for the GA, symmetry constraints were added, motivated by the fact that, modulo a phase difference, the movements of the left and right parts of the robot are symmetrical. Hence, the structure of the CPGs on the right side of the robot mirrors that of the left side. For example, the connection weight between the left hip and the left knee is equal in value to the weight connecting the right hip to the right knee. In the network the hip CPG on a given side responsible for rotation in the sagittal plane, can be connected to all the other ipsilateral[2] joint CPGs, the corresponding contralateral hip CPG, and the waist CPGs as well. Note, however, that this hip joint CPG can only receive connections from the corresponding contralateral hip joint CPG. Thus, the total number of connections to be determined sums up to 32, see also Fig. 3 for the details of inter-CPG connectivity.

For reasons that will be discussed in Sect. IV, the internal parameters of the individual two-neuron CPGs were set to fixed values, generating a frequency approximating that of a normal walking pattern. The CPG parameters were set to the following values for all CPGs, except for the knee joint CPGs and the waist joint (rotation in the sagittal plane) CPG: $\tau_u = 0.025, \tau_v = 0.3, \beta = 2.5, u_0 = 1.0, w_{12} = w_{21} = -2.0$. In analogy with human gait, the knee joint CPGs and the waist joint CPG oscillate with double frequency, compared to the other CPGs. Thus, for these joints' CPGs the $\tau_{u,v}$ values were set to half of the value for the other CPGs.

C. Genetic algorithm

A GA has been used for optimizing the structure of the CPG network controlling the movements of the robot. As mentioned above, the number of evolvable connections equals 32. In the GA, two chromosomes were used for the CPG network: one binary-valued chromosome determining the presence or absence of each of the 32 connections,

[2]The term *ipsilateral* refers to the same side of the body, and is thus the opposite of *contralateral*.

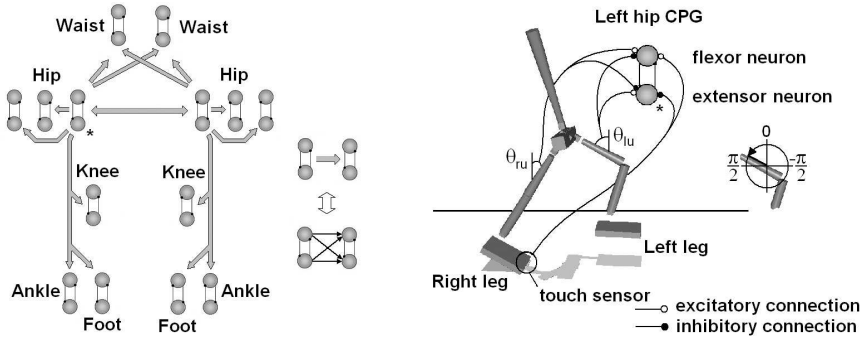


Fig. 3. Left panel: The structure of the CPG network used in the simulations. The connections are represented, in a slightly simplified way, by the arrows in the figure. Note that an arrow indicates the *possibility* of full connection, as shown in the rightmost part of the panel. Right panel: The robot depicted with a single hip joint CPG with feedback paths, and a possible choice of connection types. In the situation shown in the figure, the flexor neuron is responsible for rotating the hip joint in the counterclockwise direction.

and one real-valued chromosome determining the parameter values for those connections which are actually used in a given individual. Along with the CPG network structure, the feedback network can also be evolved using a third (real-valued) chromosome, which includes 20 parameters determining the sign and the strength of the different feedback paths (see below).

The fitness measure was normally taken as the distance walked by the robot in the initial forward direction, decreased by the sideways deviation. Some attempts were made to use a multi-objective GA (MOGA), with three populations evolving simultaneously towards different fitness measures, namely (1) the distance walked, (2) the number of times an entire foot touched the ground, and (3) the sum of the shortest distance (of the two legs) in the vertical direction between the hips and the knees over all time steps. Criterion (2) was included in order to suppress running behavior where the feet hardly touched the ground, which affected the frontal plane balance. The last criterion should promote an upright posture. However, the MOGA did not lead to any significant improvement. Hence, a standard GA was eventually chosen for the simulation experiments.

For selection, a tournament scheme of size 8 was adopted. The individuals were randomly picked from the population to compete against each other, based on the fitness values. The individual with highest fitness value was then selected with a probability equal to 0.75. After selection, the mutation operator was applied, randomly changing a gene's value with the probability $10/N$, with N being the total number of genes of the individual.

D. Feedback

In order to guide the evolutionary process towards an upright and stable bipedal gait, feedback was introduced measuring the waist angle, thigh angle, and lower leg angle, all relative to the vertical axis. Also, a touch sensor in each foot was introduced in the simulation. This sensor is used both to produce a feedback signal and to enable, or prohibit, feedback to a certain joint CPG during a specific phase, e.g. the stance phase. The feedback was incorporated into the CPGs by adding an extra term to (1), which then becomes

$$\tau_u \dot{u}_i = -u_i - \beta v_i + \sum_{j=1}^{n} w_{ij} y_j + u_0 + f \qquad (4)$$

where f is the feedback. In this setup, the feedback structure is decided upon beforehand. However, the actual type of the connection (inhibitory or excitatory) and the strength of the feedback are determined by the GA. An example of the feedback paths connected to the hip joint (and a possible choice of connection type) is shown in the right panel of Fig. 3. In detail, the feedback paths are given by the following equations:

$$\begin{aligned}
\text{waist}_1 &= c_1 w_{1_{f,e}} \theta_w + p_r[w_{1_{f,e}}(c_2\theta_{r,u} + \theta_{r,l})] \\
&\quad + p_l[w_{1_{f,e}}(c_2\theta_{l,u} + \theta_{l,r})] && (5)\\
\text{waist}_2 &= w_{2_{f,e}} \theta_{l,u} - w_{2_{f,e}} \theta_{r,u} && (6)\\
\text{hip}_{1,l} &= w_{3_{f,e}} \theta_{l,u} - w_{3_{f,e}} \theta_{r,u} + c_3 w_{3_{f,e}} e_r && (7)\\
\text{hip}_{2,l} &= e_l[w_{4_{f,e}} \theta_{l,u}] && (8)\\
\text{hip}_{3,l} &= w_{5_{f,e}} \theta_{\text{hip}_{3,l}} && (9)\\
\text{knee}_l &= e_r[w_{6_{f,e}} \theta_{r,l}] && (10)\\
\text{ankle}_l &= e_l[w_{7_{f,e}} \theta_{l,u}] && (11)\\
\text{foot}_l &= e_l[w_{8_{f,e}} \theta_{l,l}] + e_r[c_4 w_{8_{f,e}} \theta_{r,l}] && (12)\\
\text{hip}_{1,r} &= w_{3_{f,e}} \theta_{r,u} - w_{3_{f,e}} \theta_{l,u} + c_3 w_{3_{f,e}} e_l && (13)\\
\text{hip}_{2,r} &= e_r[w_{4_{f,e}} \theta_{r,u}] && (14)\\
\text{hip}_{3,r} &= w_{5_{f,e}} \theta_{\text{hip}_{3,r}} && (15)\\
\text{knee}_r &= e_l[w_{6_{f,e}} \theta_{l,l}] && (16)\\
\text{ankle}_r &= e_r[w_{7_{f,e}} \theta_{r,u}] && (17)\\
\text{foot}_r &= e_r[w_{8_{f,e}} \theta_{r,l}] + e_l[c_4 w_{8_{f,e}} \theta_{l,l}] && (18)
\end{aligned}$$

where waist_1 is the joint rotating the torso in the sagittal plane, and waist_2 denotes the joint responsible for frontal plane rotation. Likewise, hip_1 rotates the leg in the sagittal plane, while hip_2 rotates the leg in the frontal plane. The hip_3 joint is responsible for rotation around the vertical axis. The strength and the sign of the feedback paths are determined by the 16 weights $w_{i_{f,e}}$, along with the four additional constants c_i.

Since each joint CPG consists of two units, a flexor neuron and an extensor neuron, two different connection weights are used, w_{i_f} and w_{i_e}, respectively, as indicated in the equations. Apart from this, the feedback paths for the two CPG neurons are identical. Hence, the total number of parameters to be determined sums up to 20.

Furthermore, θ_w is the torso angle in the sagittal plane, $\theta_{l,u}$ is the left upper leg (thigh) angle, and $\theta_{l,l}$ is the left lower leg angle. Correspondingly, the angles for the right leg are denoted $\theta_{r,u}$ and $\theta_{r,l}$. The angle of the hip_3 joint in the local frame is denoted $\theta_{\text{hip}_{3,i}}$, where *i* is either r (right) or l (left). Finally, e_i and p_i stand for *enable* and *prohibit* respectively. If the corresponding foot is on the ground, e_i is equal to one, and zero otherwise. If the corresponding foot is *not* in contact with the ground, p_i equals one, and zero otherwise.

IV. SIMULATIONS

In this section, simulation experiments of three different methods, all involving posture-support structures, will be discussed.

In order to guide the evolution towards human-like gait, and at the same time avoid the problems related to complex fitness functions (see Sect. I), the simplest fitness measure, i.e. the distance covered, has been used here, in combination with a mass-less posture-support structure, as shown in the right panels of Fig. 2. Given a supporting structure, the robot is forced to an upright position, and only individuals capable of producing repetitive leg motion will gain high fitness. When the support is used, such individuals will appear early in the evolution. However, a drawback with this method is that when the repetitive leg motion is discovered, individuals will start to exploit the support mechanism in many different ways. One common result is that individuals tend to take unnaturally large steps. While this gives high fitness when the support is used, it will certainly have a negative effect on the frontal plane balance once the support is removed. Thus, in an attempt to avoid this motion pattern, a choice was made not to evolve the internal parameters of the individual two-neuron CPGs, shown in Fig. 1, simply because the evolution would strive towards lower frequencies. Also, in order to prevent crawling behaviors, each individual run was aborted if the hips of a robot collided with the ground.

Information concerning the simulations is given in Table I. In the following subsections the simulation experiments will be described in more detail.

A. Method 1: Four-point support

The first experiments were made using a posture support with four contact points, as shown in the right panel of Fig. 2. The four-point support was attached to the robot in such a way that there was a predetermined distance d between the contact points of the support and the ground. Different values of d were examined, as shown in the 1^{st}, 2^{nd}, and 3^{rd} rows of Table I. Feedback was not used here, and the hip_2, hip_3 and ankle joints were locked. However, no successful gait patterns were obtained in this way. The individuals simply exploited the support too much, leading to unnatural gait patterns of different kinds, also briefly described in the table. For example, the $0.02, 2$ support configuration (3^{rd} row) led to an individual performing a running gait, which one might expect to be useful, but that individual over-exploited the support to such an extent that it could not maintain its balance at all without the support. In Fig. 4, some of the resulting motions are depicted.

In order to meet the intended goal, i.e. evolving a human-like gait for the robot, two modified strategies were also tried. In the first strategy the support was gradually removed, in the sense that d increased during evolution, as better fitness values were obtained. The assumption here was that this should eventually lead to an individual that was completely independent from the support. However, this approach did not improve the outcome, compared with the previous results.

In the second strategy, the support was not gradually removed during evolution, but instead individuals were punished for using it. The fitness measure was simply decreased by a factor, properly normalized, measuring the number of ground contacts with the support. However, this approach did not yield any improvements either.

In the case of four-point support no useful results were obtained; the individuals simply exploited the support too much, resulting in unnatural gait patterns. For example, when using the first modified strategy, gradually removing the support, a slow unstable gait pattern, resembling a drunkard's walk, emerged. When using the second modified strategy, with punishment for support usage, the result was an individual using a hop gait for locomotion.

B. Method 2: Two-point support in 2D, then 3D

Since no acceptable results were found with the four-point support, the support structure was changed to one having only a single contact point on each side of the robot, as seen in the rightmost panel of Fig. 2. Rather than evolving 3D balance at once, as in the previous case, the idea now was to divide the problem into two phases; first evolving gait in 2D, and second, to generalize it to a full 3D gait.

In this procedure, a CPG network capable of producing a stable upright gait in the sagittal plane should first be evolved using the two-point support, with the hip_2, hip_3 and ankle joints locked at this stage. Second, when a stable individual has been obtained, it should be cloned creating a new population consisting of copies of this individual. At this stage, the support should be removed and the GA should find a way to balance the robot in the frontal plane as well. Before the evolution starts, the hip_2 and ankle joints should be unlocked and the corresponding genes, including the genes encoding the waist joint parameters, should be randomly initiated for each individual in the population. Since the remaining genes (which are identical for all individuals) ensure sagittal plane balance they should not be changed in the second step.

1) Phase 1: Evolving balance in the sagittal plane: During this first phase the hip_2, hip_3 and ankle joints were locked. Balance in the sagittal plane was evolved using a

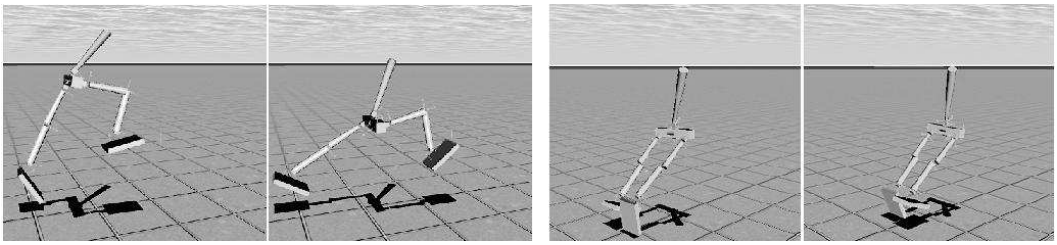


Fig. 4. The left panel shows the simulated robot taking unnaturally large steps. The right panel shows the robot exploiting the four-point support. Note that the supporting structure is not shown in the pictures.

two-point support, with contact points placed 2 m from the robot and 0.25 m above the ground. This configuration was chosen since it ensures low sideways leaning angle and at the same time allows the robot to bend its knees without the support touching the ground. Furthermore, if a robot's hips collided with the ground, the evaluation of that particular individual was terminated.

In order to balance in the sagittal plane the evolutionary procedure started misusing the torso as a third leg, achieving speeds up to 0.3 m/s. This problem was solved by simply removing the contact point in the torso which is used to detect the collision with the ground. Once the torso could not be used for support, evolution found the large step motion, as described earlier, ensuring balance in the sagittal plane with a speed of approximately 0.45 m/s, see the 4^{th} and 5^{th} rows of Table I.

In order to reduce the step length, hand-tuned feedback paths were introduced measuring torso, upper leg, and lower leg angles, as described in Sect. III. Adding feedback, a significant reduction in evolution time was observed. The same fitness value as before could now be reached approximately 5 times faster. However, the individuals were still taking unnaturally large steps.

Success in obtaining an upright human-like gait, with normal step size, was achieved using the following rule: during the evaluation of each individual, if the robot's hip fell below a certain value, the support was removed until the end of that run. If the step length is large and the support is removed in this way, the robot will most likely be unable to maintain the frontal plane balance. Thus, it will fall to the ground ending the run. Forced by this rule, evolution was able to find individuals moving at a speed of 1.13 m/s, see Table I, 6^{th} row. However, even after 400 generations, these individuals could not walk more than 10 to 15 meters before falling to the ground. In order to improve the performance, the GA was allowed to evolve the feedback paths as well. As a result, a stable individual, i.e. one that did not fall even after the end of the nominal evaluation time, was obtained within 50 generations, walking at a speed of 0.4 m/s, see Table I, 7^{th} row. To ensure stability, the whole foot sole was on the ground during almost the entire stance phase, resulting in a perfect condition for full 3D balance.

2) Phase 2: Evolving balance in full 3D: Once a satisfactory stable individual had been obtained using the support, evolution in the full 3D environment could begin. In the initial population at this stage, all individuals were mutated copies of the best individual from the previous step, as described above. The GA should now only consider the hip_2, ankle, and waist joints, as well as their feedback paths. The fitness measure was still taken as the distance covered in the initial forward direction, decreased by the sideways distance.

TABLE I

PARAMETERS AND RESULTS OF THE SIMULATION RUNS

In the column labeled *support*, the numbers i, j denote the initial placement of the contact points in a given run, where i is the height above the ground [m], and j is the horizontal distance from the hip [m]. Evaluation time is denoted by t [s], and the f column indicates whether or not feedback was used. F [m] is the obtained fitness, v is the average locomotion speed [m/s] of the robot during the evaluation period, and the last column gives a short description of the resulting gait. Note: † denotes phase 1, and ‡ denotes phase 2.

Support	t	f	F	v	Gait
4-point, 0.3, 2	7	No	3.85	0.55	hop gait
4-point, 0.1, 1	7	No	4.60	0.66	large steps
4-point, 0.02, 2	7	No	6.55	0.93	running
2-point, 0.25, 2 †	7	No	2.10	0.30	tripod gait
2-point, 0.25, 2 †	7	No	3.15	0.45	large steps
2-point, 0.25, 2 †	7	Yes	7.91	1.13	running
2-point, 0.25, 2 †	20	Yes	7.21	0.38	slow, stable
no support ‡	40	Yes	18.26	0.46	slow, stable
1 sec. 0.25, 2 †	40	Yes	19.54	0.56	slow, stable
1 sec. 0.25, 2 ‡	40	Yes	23.09	0.58	slow, stable
1 sec. 0.25, 2	40	Yes	35.56	0.90	fast walk

Within 150 generations, the best individual was able to maintain balanced walking for up to 60 seconds. After an additional 100 generations, the best individual was generally able to maintain balance indefinitely, see Table I, 8^{th} row. Since the robot was unaware of its direction of motion and because of the fact that the hip_3 joints were locked, the smallest perturbation would set it out of course, resulting in a lower fitness value.

Given the best individual from phase 2, it was possible to continue evolving the parameters for the hip_3 joint. The feedback for the hip_3 joint was defined as described in Sect. III. Evolution found a solution (not shown in the table) striving towards keeping the feet facing forward. A similar gait as before emerged.

C. Method 3: 2D one second support, then 3D

The reason for introducing the two-point posture-support structure in Method 2 was mainly that it is much harder

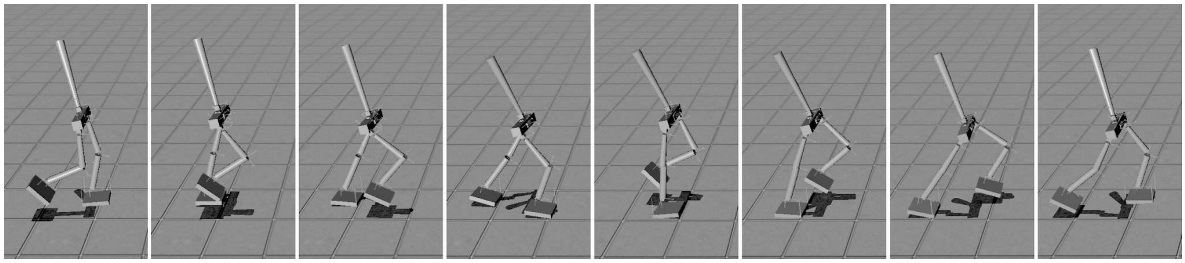


Fig. 5. The best evolved stable gait pattern in the full 3D environment. The details of the corresponding individual are shown on the 10^{th} row of Table I.

to maintain balance in the frontal plane than in the sagittal plane. By using the supporting structure, the problem was separated into two stages of evolution; first, generating a stable gait in 2D, and second, generalizing the 2D gait to three dimensions. The assumption here was that this way of splitting up the problem should make it easier for evolution to find a good solution to the overall problem of generating a robust 3D gait. However, since the hip_2 and ankle joints were locked during the 2D stage the balance in the frontal plane is then only ensured by the torso. As a consequence, evolution most often creates individuals that solve the problem in an unnatural way, i.e. individuals that exploited the supporting structure too much. Such individuals are usually not suitable for further evolution in 3D. Another drawback of Method 2 is that there is no obvious way of deciding at what point to interrupt the 2D evolution stage, and thus to enter the 3D phase: This simply has to be judged by the experimenter in an *ad hoc* manner. Hence, there is no guarantee that the individuals evolved in 2D will perform well in the 3D stage.

Therefore, a third method was investigated as well. The same two-point supporting structure as described in the 2D case in the previous method, was used. The difference here, as compared to the other method, is that under the first stage in 2D the individuals were evaluated in a procedure where the supporting structure was present only during the first second of the evaluation, and was then completely removed. This arrangement is motivated by the fact that it is during the start sequence, before entering the gait cycle, that the individuals are most vulnerable, in terms of frontal plane balance. Here, the hip_2, hip_3 and ankle joints were still locked. However, after some generations most individuals in the population should be able to walk without the supporting structure present, except during the initial second. This has been confirmed to work well in simulation experiments. The goal in this stage was to obtain a large portion of individuals able to walk in cautious manner, which is essential for individuals to be able to generalize to 3D. In the next stage, evaluations were performed in full 3D. That is, the evaluation procedure was performed in the same way as described above, but all joints except the hip_3 joints were unlocked. The 9^{th} row of Table I shows the results from evolution in 2D (joints locked and 1 second support), and the 10^{th} row of the table shows the results from phase 2 (evolution based on the best individual from the previous run, with hip_2 and ankle joints unlocked). A visualization of the gait for the best individual, corresponing to the 10^{th} row in the table, is shown in Fig. 5.

Moreover, the results from a run with hip_2 and ankle joints unlocked, performed in a single step, i.e. without the two-phase procedure described above, is shown in the last row. The gait obtained in this last run was fast, but appeared to be rather unstable.

V. DISCUSSION AND CONCLUSIONS

The outcome of the examinations and experiments described in this paper indeed fulfilled the intended goal, i.e. to generate robust bipedal gaits for a simulated robot by means of structural evolution of CPG networks: Two of the three methods introduced in this paper solved the problem of generating gaits for the simulated bipedal robot in 2D and 3D environments. However, Method 2 was affected by some drawbacks, compared to the third method. Firstly, since the support structure is present the whole time during phase 1 (evolution in 2D), evolution might very well find solutions that receive high fitness scores in 2D, but are less successful in generalizing to 3D. Examples of such solutions are individuals that walk with too long steps, giving high fitness in 2D because of their ability to cover large walking distances in short time. However, this kind of walking behavior seriously affects the frontal plane balance in 3D. Thus, evolution in 2D has to be aborted at an appropriate time, before this kind of behavior emerges which, in turn, requires that the evolutionary process is monitored more or less continuously in order to determine when it should be interrupted.

Secondly, while it is possible (at least in principle) to monitor the progress and stop the evolution when sufficient locomotion speed is achieved, it is not always the case that evolution chooses a path, i.e. relatively small steps with a high speed, that is suitable for further evolution in 3D. Often the large step motion behavior emerges before any stable, small-step gait pattern is obtained.

In the case of the third method the two problems described above are not present: Evolution is biased towards generating gaits capable of handling the 3D environment from the beginning. Thus, Method 3 seems to be the most promising candidate for future investigations.

The need for the support structure during the initial second, as described in the second method, indicates that the CPG network cannot fully handle the start-up of the walking cycle in an appropriate way. Thus, one should, in future work, consider a dedicated controller, either a CPG-based

controller or some other type of controller, for the start-up sequence of the walking cycle. It should then be tuned to enter the walking cycle and hand over to a CPG network in a more smooth way. Then, ultimately, it would be possible to skip totally the support structure.

In this paper only straight-line walking has been considered, i.e. no turning motions were involved. However, one could include such motions using the hip_3 joint in order to change deliberately the direction of walking, preferably by using vision for feedback.

Another topic for future work would be to investigate whether one could evolve the over-all feedback network, without having to pre-specify certain feedback paths, as is currently done. However, such an approach would probably increase the evaluation time considerably, since the likelihood of finding a set of feedback paths in an early generation that generates any gait at all would most probably be very small. On the other hand, evolving the feedback network could lead to better overall performance, compared to specifying the paths *ad hoc*.

REFERENCES

[1] R. A. Brooks, C. Breazeal, M. Marjanovic, and B. Scassellati, "The cog project: Building a humanoid robot," *Computation for Metaphors, Analogy, and Agents*, vol. 1562, pp. 52–87, 1999.

[2] H. Kozima and J. Zlatev, "An epigenetic approach to human-robot communication," in *Proc 9th Int Workshop on Robot and Human Interactive Communication (RO-MAN'00)*, Paris, France, 23–26 Mar. 2000, conference.

[3] T. Minato, M. Shimada, H. Ishiguro, and S. Itakura, "Development of an android robot for studying human-robot interaction." in *Proc 17th Int Conf on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2004*, 2004, pp. 424–434.

[4] A. Takanishi, M. Ishida, Y. Yamazaki, and I. Kato, "The realization of dynamic walking by the biped walking robot WL-10RD," in *Proc Int Conf on Advanced Robotics (ICAR'85)*, 1985, pp. 459–466.

[5] K. Hirai, M. Hirose, Y. Haikawa, and T. Takenaka, "The development of honda humanoid robot," in *Proc Int Conf on Robotics and Automation (ICRA'98)*. IEEE, 1998, pp. 1321–1326.

[6] T. Arakawa and T. Fukuda, "Natural motion generation of biped locomotion robot using hierarchical trajectory generation method consisting of GA, EP layers," in *Proc Int Conf on Robotics and Automation (ICRA'97)*. IEEE, 1997, pp. 211–216.

[7] A. L. Kun and W. T. Miller, "Control of variable speed gaits for a biped robot," *IEEE Robotics & Automation Magazine*, vol. 6, no. 3, pp. 19–29, Sep 1999.

[8] H. Wang, T. T. Lee, and W. A. Gruver, "A neuromorphic controller for a three-link biped robot," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, no. 1, pp. 164–169, Jan/Feb 1992.

[9] G. Taga, Y. Yamaguchi, and H. Shimizu, "Self-organized control of bipedal locomotion by neural oscillators in unpredictable environment," *Biological Cybernetics*, vol. 65, pp. 147–159, 1991.

[10] J. Pettersson, H. Sandholt, and M. Wahde, "A flexible evolutionary method for the generation and implementation of behaviors for humanoid robots," in *Proc 2nd Int Conf on Humanoid Robots (Humanoids'01)*, IEEE-RAS, Waseda University. Tokyo, Japan: Humanoid Robotics Institute, 22-24 Nov. 2001, pp. 279–286.

[11] J. Shan, C. Junshi, and C. Jiapin, "Design of central pattern generator for humanoid robot walking based on multi-objective ga," in *Proc Int Conf on Intelligent Robots and Systems (IROS 2000).*, vol. 3. Takamatsu, Japan: IEEE-RSJ, 2000, conference, pp. 1930–1935.

[12] M. Y. Cheng and C. S. Lin, "Genetic algorithm for control design of biped locomotion," *Journ. of Robotic Systems*, vol. 14, no. 5, pp. 365–373, 1997.

[13] K. Wolff and P. Nordin, "Learning biped locomotion from first principles on a simulated humanoid robot using linear genetic programming," in *Proc Genetic and Evolutionary Computation Conf (GECCO'03)*, ser. LNCS, E. Cantú-Paz, Ed., vol. 2723, AAAI. Chicago: Springer Verlag, 12-16 July 2003, pp. 495–506.

[14] J. Ziegler, J. Barnholt, J. Busch, and W. Banzhaf, "Automatic evolution of control programs for a small humanoid walking robot," in *Proc 5th Int Conf on Climbing and Walking Robots (CLAWAR'02)*, P. Bidaud, Ed. Professional Engineering Publishing, 2002, pp. 109–116.

[15] S. Grillner, "Neural networks for vertebrate locomotion." *Scientific American*, vol. 274, pp. 64–69, 1996.

[16] S. Grillner, T. Deliagina, Ö. Ekeberg, A. El Manira, R. Hill, A. Lansner, G. Orlovsky, and P. Wallen, "Neural networks that coordinate locomotion and body orientation in lamprey," *Trends in Neurosciences*, vol. 18, no. 6, pp. 270–279, 1995.

[17] Ö. Ekeberg, "A combined neuronal and mechanical model of fish swimming," *Biological Cybernetics*, vol. 69, no. 5-6, pp. 363–374, oct 1993.

[18] P. Wallén, Ö. Ekeberg, A. Lansner, L. Brodin, H. Tråvén, and S. Grillner, "A computer-based model for realistic simulations of neural networks. II: The segmental network generating locomotor rhythmicity in the lamprey," *J. Neurophysiol.*, vol. 68, pp. 1939–1950, 1992.

[19] T. G. Brown, "The factors in rhythmic activity of the nervous system." *Proc R Soc London Ser*, vol. 85, pp. 278–289, 1912.

[20] S. Grillner and P. Zangger, "The effect of dorsal root transection on the efferent motor pattern in the cat's hindlimb during locomotion." *Acta Physiol Scand*, vol. 120, pp. 393–405, 1984.

[21] J. Duysens and H. W. A. A. V. de Crommert, "Neural control of locomotion; part 1: The central pattern generator from cats to humans." *Gait and Posture*, vol. 7, no. 2, pp. 131–141, 1998.

[22] G. Taga, "Nonlinear dynamics of the human motor control - real-time and anticipatory adaptation of locomotion and development of movements," in *Proc 1st Int Symp on Adaptive Motion of Animals and Machines (AMAM'00)*, 8–12 Aug. 2000.

[23] T. Reil and P. Husbands, "Evolution of central pattern generators for bipedal walking in a real-time physics environment." *IEEE Transactions in Evolutionary Computation*, vol. 6, no. 2, pp. 159–168, 2002.

[24] H. Kimura, S. Akiyama, and K. Sakurama, "Realization of dynamic walking and running of the quadruped using neural oscillator," *Autonomous Robots*, vol. 7, no. 3, pp. 247–258, 1999.

[25] K. Tsuchiya, S. Aoi, and K. Tsujita, "Locomotion control of a biped locomotion robot using nonlinear oscillators," in *Proc Int Conf on Intelligent Robots and Systems (IROS'03)*. IEEE/RSJ, 2003, pp. 1745–1750.

[26] M. Lewis, F. Tenore, and R. Etienne-Cummings, "CPG design using inhibitory networks," in *Proc Int Conf on Robotics and Automation (ICRA'05)*, IEEE-RAS. Barcelona, Spain: Wiley, 18-22 Apr. 2005.

[27] M. Ogino, Y. Katoh, M. Aono, M. Asada, and K. Hosoda, "Reinforcement learning of humanoid rhythmic walking parameters based on visual information," *Advanced Robotics*, vol. 18, no. 7, pp. 677–697, 2004.

[28] K. Matsuoka, "Mechanisms of frequency and pattern control in the neural rhythm generators." *Biol Cybern*, vol. 56, no. 5–6, pp. 345–353, 1987.

[29] C. Paul and J. Bongard, "The road less travelled: Morphology in the optimization of biped robot locomotion," in *Proc Int Conf on Intelligent Robots and Systems (IROS'01)*, vol. 1. Maui, HI, USA: IEEE/RSJ, 2001, pp. 226–232.

[30] G. M. Shepherd, *Neurobiology*, 3rd ed. Oxford University Press, 1994, ch. 20, pp. 435–451.

[31] T. G. Brown, "The intrinsic factors in the act of progression in the mammal." *Proc R Soc London Ser*, vol. 84, pp. 308–319, 1911.

[32] E. Ott, *Chaos in Dynamical Systems*. New York: Cambridge University Press, 1993.

[33] J. Pettersson, "EvoDyn: A simulation library for behavior-based robotics," Department of Machine and vehicle systems, Chalmers University of Technology, Göteborg, Technical Report, September 2003.

[34] R. Featherstone, *Robot Dynamics Algorithms*. Kluwer Academic Publishers, 1987.

Paper V

UFLibrary: A Simulation Library Implementing the Utility Function Method for Behavioral Organization in Autonomous Robots

submitted to

International Journal on Artificial Intelligence Tools, August 2005.

International Journal on Artificial Intelligence Tools

UFLIBRARY: A SIMULATION LIBRARY IMPLEMENTING THE UTILITY FUNCTION METHOD FOR BEHAVIORAL ORGANIZATION IN AUTONOMOUS ROBOTS

Jimmy Pettersson and Mattias Wahde

Department of Applied Mechanics
Chalmers University of Technology
412 96 Göteborg
Sweden
{*jimmy.pettersson, mattias.wahde*}*@chalmers.se*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

A simulation software package (UFLibrary) implementing the utility function (UF) method for behavioral selection in autonomous robots, is introduced and described by means of an example involving a simple exploration robot equipped with a repertoire of five different behaviors. The UFLibrary (as indeed the UF method itself) is aimed at providing a rapid yet reliable and generally applicable procedure for generating behavioral selection systems for autonomous robots, while at the same time minimizing the amount of hand-coding related to the activation of behaviors.

It is demonstrated how the UFLibrary allows a user rapidly to implement individual behaviors and to set up and carry out simulations of a robot in its arena, in order to generate and optimize, by means of an evolutionary algorithm, the behavioral selection system of the robot.

Keywords: Behavior-based robotics; utility function; behavioral organization; autonomous robots, evolutionary robotics.

1. Introduction

The research field of behavior-based robotics (BBR)[1] is concerned with building robotic control systems (hereafter referred to as (*robotic*) *brains*, to avoid confusion with the more limited systems based on classical control theory) in a bottom-up fashion, starting from simple behaviors and combining these to form a robotic brain capable of complex overall behavior. As opposed to classical artificial intelligence (AI)[2], BBR has not been strongly focused on human-level reasoning and cognition, and has taken a more generous definition of intelligence, inspired by the ability to survive and reproduce exhibited even by simple life forms. In implementations, BBR-based subsystems are generally capable of handling low-level tasks such as e.g. obstacle avoidance and wall-following, whereas methods from classical AI are often used in subsystems handling higher-level behaviors. However, reconciling two

such different approaches (BBR and classical AI) to form the brain of a robot is, at best, a temporary solution. In the authors' view (based on the capabilities of biological organisms, as generated by evolution), the road to truly intelligent machines is more likely to come from an extension of the behavior-based approach to form more complex robotic brains than those considered so far.

Reaching such a goal, however, is a formidable problem involving several complex tasks. So far, BBR has been successful in generating robots capable of simple tasks, reaching approximately (some aspects of) insect-level intelligence. Yet the elusive ultimate goal of robotics research is, of course, to generate machines capable of human-level intelligence (or beyond), which can serve a useful purpose in society. One of the main obstacles involved in extending the BBR-approach to higher levels of complexity is the problem of behavioral organization, also known as action selection or behavioral selection[3,4], which will now be described briefly.

1.1. *Behavioral Organization*

Simplifying somewhat, a robotic brain in BBR can be considered as the conjunction of (1) a repertoire of behaviors and (2) a method for selecting which behavior to activate in any given situation (see Fig.1). The problem of behavioral selection has been studied by many authors (for a review, see e.g. Ref. 3), and a variety of methods have been suggested. In arbitration methods, i.e. the kind that will be studied in this paper, one behavior is given control of the robot, even though the *identity* of that behavior of course will vary with time, whereas in behavior fusion methods, the action taken by the robot is a weighted average of the suggestions from many behaviors.

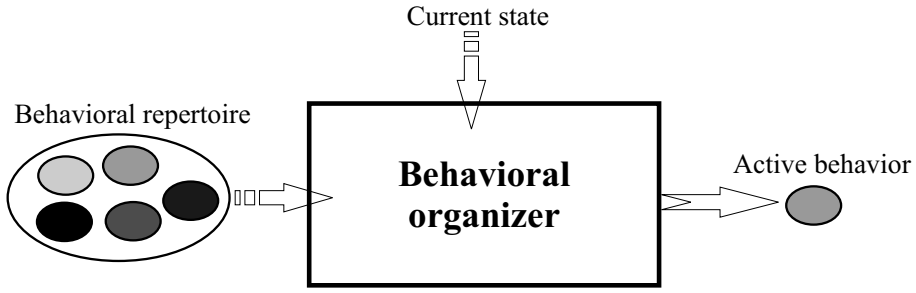


Fig. 1. Illustration of the role of a behavioral organizer – selection of the most appropriate behavior based on the current state of the robot and its environment.

A common problem in most methods for behavioral selection has been the fact that they require the user to specify, in detail, the parameters determining the selection and activation of behaviors (see Ref. 5 and 6), and thus effectively to judge the relative importance of different behaviors in all situations encountered by the robot, something which is notoriously difficult to do. By contrast, one of the

main purposes of the utility function (UF) method[4] is to delegate the details of the behavioral selection procedure to a structure based on utility functions that, in turn, are generated by means of an evolutionary algorithm (EA).

An additional aim with the introduction of the UF method has been to provide a *generally applicable* method for behavioral selection, i.e. one that is as independent as possible of the details of the particular problem being considered, and also independent of the specific implementations used for the constituent behaviors. For illustration purposes, a specific example, namely a simple exploration robot, will be presented in this paper. However, the UF method has also recently been applied to other problems[7,8].

The utility functions, which will be described further in Sect. 2 below, serve the purpose of providing a common currency for assessing the relative importance of different behaviors in any given situation. The notion of utility has a long history in fields such as economic theory and ethology[9], and has also been used in connection with autonomous robots[10,11]. However, the utility function method represents, to the authors' knowledge, the first attempt at defining a general-purpose method for behavioral organization based on evolutionary optimization of utility functions.

1.2. *Software*

In addition to a method for behavioral selection, a software library for its implementation is also needed, particularly in a method such as the UF method which relies on an EA, commonly requiring simulations to be carried out before the robotic brain is transferred to an actual robot.

While there are several software libraries implementing the physical aspects of autonomous robots (such as the equations of motion, collision handling etc.), see e.g. Ref. 12–16, there are, to the authors' knowledge, no publicly available, general-purpose libraries implementing a specific method for behavioral selection. The main aim of this paper is to introduce and describe such a library (UFLibrary) and to illustrate how it can be used rapidly to evolve a behavioral organization system for any given, complex task.

2. The Utility Function (UF) Method

2.1. *Basic description*

The utility function (UF) method is a biologically inspired arbitration method[4] for behavioral organization (selection) in autonomous robots. Here, only a brief description of the UF method will be given. For a more detailed introduction, see e.g. Ref. 4 or Ref. 17. In the UF method, each individual behavior is associated with a utility function, and the selection (for activation) of behaviors is based on the values of the utility functions, in such a way that *the behavior with maximum current utility is active*. In general, a utility function U_i, corresponding to behavior

i, can be written as

$$U_i = U_i(\mathbf{s}, \mathbf{p}, \mathbf{x}) \ , \tag{1}$$

where $\mathbf{z} = (\mathbf{s}, \mathbf{p}, \mathbf{x}) = (s_1, s_2, \ldots, p_1, p_2, \ldots, x_1, x_2, \ldots)$ are the state variables, divided into three categories: external variables ($\mathbf{s}$), such as e.g. the readings of proximity sensors, internal physical variables ($\mathbf{p}$), such as e.g. the readings of a sensor measuring the current battery level, and internal abstract variables ($\mathbf{x}$), which are the readings of internal signaling variables referred to as hormones, in keeping with the biological analogy of the UF method. In principle, each utility function may depend on the full set $\mathbf{z} = (\mathbf{s}, \mathbf{p}, \mathbf{x})$ of state variables. However, in practice, most utility functions will only depend directly on a few components of $\mathbf{z}$.

In order to generate a behavioral organizer for a given robot, the utility functions must be generated. In addition, the functions describing the variation of hormone levels (hereafter: hormone functions) in each behavior, must be specified, as must the fitness function (see below). Once the utility functions and the hormone functions are available, behavioral selection is straightforward: At any given time, the variables $\mathbf{s}$ and $\mathbf{p}$ can be measured and $\mathbf{x}$ can be computed, after which the values of the utility functions U_i can be obtained, for each behavior i, and the active behavior i_{active} can be obtained as

$$i_{\text{active}} = \arg\max(U_i) \ , \quad i = 1, \ldots, N, \tag{2}$$

where N is the number of behaviors. The problem, of course, is how to specify the utility functions. In the UF method, these functions are normally obtained using an evolutionary algorithm (EA), even though there is nothing preventing the user of the method to set these functions by hand, should it be possible to do so. In principle, the functions may take any form. However, in practice, a polynomial ansatz is often used for each function. Thus, for example, for a utility function $U = U(s_1, x_1)$ the following ansatz could be made:

$$U = a_{00} + a_{10}s_1 + a_{01}x_1 + a_{20}s_1^2 + a_{11}s_1x_1 + a_{02}x_1^2 \ , \tag{3}$$

where the a_{ij} are constants to be determined by the EA. Thus, when running the UF method (e.g. by using the UFLibrary described here), the relevant state variables and the polynomial degree is specified for each utility function, and the EA then proceeds to optimize the parameters in these functions, so as to achieve the desired overall behavior, the specification of which is given by the fitness function used in the EA, which will now be described.

2.1.1. *Fitness functions*

In the UF method, behaviors are divided into two categories, *task behaviors* (which are associated with a non-zero fitness function) and *auxiliary behaviors* (which do not affect the fitness of the robot). Thus, a user of the method need only specify the fitness function for the task behavior(s). How, then, can the auxiliary behaviors

be activated? Consider a simple case of a robot equipped with two behaviors, a task behavior (B1) for floor-sweeping, and an auxiliary behavior (B2) for battery charging. Only B1 gives a fitness increase, and an evolved robot will thus strive to use B1 as much as possible (the fitness function can e.g. be taken as the fraction of time spent in B1, or the area covered by the robot while executing B1). However, from time to time, the robot *must* activate B2 in order not to run out of energy. Thus, if properly evolved, the utility functions of the robot will be such that the utility of B2 will rise as the battery energy falls, eventually surpassing the utility of B1 and thus activating B2. As the battery becomes charged, the utility of B2 will gradually fall, so that B1 again can be activated etc.

Hence, the evolved utility functions will take care of the activation of behaviors at appropriate times, thus freeing the user from the daunting task of specifying the relative importance of all behaviors. Indeed, this is one of the main advantages of the UF method[4]: In many cases, there will be only one task behavior, for which the fitness function often can be specified quite easily.

2.2. *Hormone functions*

In addition to the utility functions and the fitness function, the hormone functions must also be specified. Thus, for example, a visual detection of an object in front of a robot, a situation which would evoke an emotion akin to fear, may raise the level of a hormone H, which could raise, say, the utility of a behavior for obstacle avoidance or fleeing etc., The raise in hormone level may remain for some time, thus (for example) keeping the obstacle avoidance behavior active for some time even after the direct sensory input has vanished e.g. as a result of a change of direction of the robot.

Ideally, in order to allow maximum flexibility, the hormone functions should be *evolved* just like the utility functions. In the general case, those functions would depend on the state variables, and the exact variation (i.e. the polynomial coefficients) would be determined by the EA, as in the case of the utility functions. However, this feature has yet to be implemented in the UFLibrary. Thus, at present, the hormone functions must be specified by the user. In this paper, only very simple hormone functions will be used, in which the levels of hormones are either 1 or 0, as described in Sect. 4.1 below.

2.3. *Hierarchical levels of behaviors*

The above description of the UF method (as indeed the description given in Ref. 4), is somewhat simplified, since it neglects the concept of hierarchical levels of behaviors, which will now be described briefly.

Since the UF method tries to keep individual behaviors as simple as possible, it is often so that a given behavior may be divided into several sub-behaviors which are easier to specify than the complete behavior (in effect, this amounts to placing a heavier burden on the EA evolving the utility functions, while simplifying for

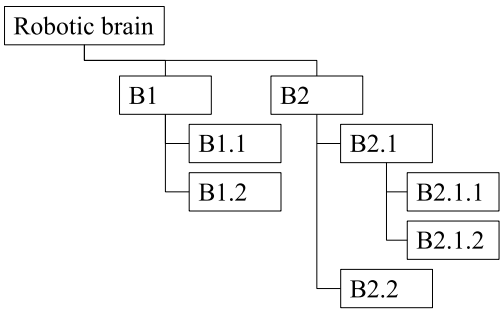


Fig. 2. An illustration of hierarchical levels of behaviors.

the user of the method). In such cases, a level-by-level, tree-like comparison is made in order to find the active behavior, as illustrated in Fig. 1. In this example, eight behaviors are available, namely B1, B1.1, B1.2, B2, B2.1, B2.1.1, B2.1.2, and B2.2. When the robot starts its operation, it will first compute the values of all the utility functions, and then compare the utilities U_1 and U_2 of B1 and B2, respectively. Consider the case where $U_2 > U_1$. In that case, a behavior from the branch emanating from B2 will be selected, beginning with a comparison of $U_{2.1}$ and $U_{2.2}$. If, say, $U_{2.1} > U_{2.2}$, a behavior from the branch emanating from B2.1 will be selected. Thus, $U_{2.1.1}$ and $U_{2.1.2}$ are compared, and the behavior with the highest utility (B2.1.2, say) is activated. As the robot continues its operation, a level-by-level comparison is performed at all time steps. As long as $U_2 > U_1$, $U_{2.1} > U_{2.2}$, and $U_{2.1.2} > U_{2.1.1}$, B2.1.2 will be active. If say, $U_{2.1.1}$ suddenly exceeds $U_{2.1.2}$, the behavior B2.1.1 will be activated instead of B2.1.2. On the other hand if B1 suddenly exceeds B2, a behavior from the branch beginning at B1 will be selected for activation, namely B1.1 or B1.2, depending on which of these two behaviors is associated with the highest current utility value.

In addition to functioning as internal signals, hormone variables (or, more exactly, their corresponding internal abstract state variables $\mathbf{x}$) may also prevent dithering between behaviors, by instantaneously raising the utility of an activated behavioral hierarchy (providing, of course, that the utility functions have evolved to do so). For example, in Fig. 2, if B1 corresponds to some task behavior (e.g. floor-sweeping) and B2 is a battery charging behavior consisting of several sub-behaviors (e.g. finding a charging station, approaching a charging station etc.), the activation of a behavior, e.g. B2.1.1, in the hierarchy beginning at B2 should preferably raise the utility of B2 so that U_2 will remain above U_1 for some time. However, it is B2.1.1 rather than B2 which is activated, so how can U_2 be raised? This is done by calling an *exit* procedure for each level in the hierarchy that is being deactivated, and an *entry* procedure for each behavior in the hierarchy that is being activated. Thus, in switching from, say, B1.1 to B2.1.1, the exit procedures of both B1.1 and B1 are called, and the entry procedures for B2, B2.1 and B2.1.1 are called. Both

the entry and the exit procedures may modify hormone levels.

Thus, in the example just described, a hormone H corresponding to *hunger* may be responsible for activating the hierarchy under B2. However, as the robot begins charging the batteries, (the level of) H would decrease, thus activating the B1 hierarchy, again raising the hunger level etc. To prevent such dithering, the activation of the B2 hierarchy may be associated with an instantaneous rise in H, which, in turn, may raise the utility U_2 of behavior B2, thus preventing immediate re-activation of B1.

3. The UFLibrary

The UFLibrary has been written using the Borland Delphi object-oriented Pascal language[18]. UFLibrary makes possible the construction of robotic brains of arbitrary complexity, provided that certain guidelines (as enforced by the UF method) are followed. The functionality is provided in the form of a simulation library, similar to a dynamic link library (DLL), rather than as a stand-alone application. Note that the UF method (and thus the UFLibrary) is concerned with behavioral *organization*, i.e. the process of selection of appropriate behaviors at all times, rather than the actual generation of the constituent behaviors included in the behavioral repertoire. Thus, the behaviors in the repertoire can be generated by whatever means chosen by the user who, for example, may generate some behaviors by hand, and evolve other behaviors using any suitable software, before including them in the repertoire as described in Sect. 3.2 below.

Although some programming is required to make full use of the UFLibrary (for example to define new behaviors or to define a new stand-alone application based on the library), it is possible, for basic usage, to control most aspects of the simulations through the use of simple text files. These files have an object oriented structure (see Fig. 5) and provide an informative and intuitive interface, even to users that have no programming experience.

When implementing an application, a minimum of two definition files are required; one for the world (arena) in which the robot is supposed to move and operate, and one for the robot itself. Each file contains the definition of entities known as *objects*, as well as the values of their respective parameters. Whereas the robot definition file, an example of which is given in Ref. 17, often contains a hierarchical structure with the body and the brain being on the highest level, the arena definition file consists of a simple list of objects such as walls, doorways, furniture, and windows.

The contents and usage of the UFLibrary will now be introduced. A summary of the usage procedure is given in Sect. 3.3 below.

3.1. *Contents of the UFLibrary*

The UFLibrary contains a total of more than $12,000$ lines of code, divided into 51 source units, ranging from general units concerning e.g. the evolutionary algorithm

(EA) to more specific units such as those defining, for example, a DC motor or a laser range finder. In addition, a few basic behaviors, such as a navigation behavior, an obstacle avoidance behavior etc., have been included as well.

The library contains a general implementation of an EA that, by default, uses generational replacement, tournament selection, elitism, single point crossover, and both creep and full range mutations, even though other operators, such as, for example, roulette-wheel selection and steady-state replacement, are available as well. The task of the EA is to optimize the utility functions responsible for the selection of behaviors. In the basic implementation, each utility function is considered as *one* gene and, by default, the crossover procedure works by cutting the genome at a crossover point between two genes, and swapping the genes downstream from the crossover point between individuals. However, a user may override this procedure in order to implement a custom crossover procedure. Mutations modify the coefficients of the polynomials defining the utility functions. In this case as well, a user may override the mutation procedure to allow e.g. the addition or removal of polynomial terms.

The EA operates on a population of agents, where an agent is an abstract base object that only provides access to a genome and a fitness value. For the evolution of agents with a physical appearance, the UFLibrary includes a robot class that extends the functionality of an agent by including physical properties, such as a body and a brain. Both the body and the brain are instantiated through text files written by the user. This allows the user to incorporate any type of robot into the simulation, provided that the correct interface is implemented in the derived classes.

As indicated above, the library also contains several objects that are not directly related to the UF method but are crucial for the simulation of a robot, for example a dynamical model of a differentially steered robot, and a model of a DC motor (TDCMotor), derived from the base class TMotor. When defining a robot, two such motors (one for each wheel) can be added to the body of the robot, through its definition file, an example of which can be found in Ref. 17. Alternatively, the user may write a custom motor class, derived from TMotor. In addition, several sensors types, namely a 2D laser range finder, a battery sensor, and an IR detector, have been defined, and can be added to a robot through its definition file. IR detectors read the signals from an IR beacon, an implementation of which is also available in the UFLibrary.

Simplified (2D) collision checking is also provided, and operates by slicing the arena in horizontal sections and checking for line intersection between the slice of the arena and the geometric shape of the robot. This simplified collision checking can be used by individual behaviors and also as a possible termination criterion for the evaluation of a robot.

3.2. *Defining and adding behaviors*

The addition of a behavior to the behavioral repertoire contained in a robotic brain is a two-step procedure. First, the user must write the actual behavior class (unless the behavior already exists, and can be used in an off-the-shelf manner). Second, the behavior must be added to the definition file describing the structure of the robotic brain.

3.2.1. *Writing a behavior class*

When implementing a new behavior, together with its corresponding class, there are a number of rules, enforced by the base class TBehavior (included in the UFLibrary) that must be followed. The TBehavior class has been defined specifically so as to minimize the amount of work carried out when defining a new behavior. Thus, all new behaviors are derived from the base class TBehavior, and the specifics of the behavior in question must be provided by writing a few procedures overriding the corresponding procedures in the base class, namely the *Step* procedure defining the action(s) taken in each time step, the *Enter* and *Exit* procedures (described in Sect. 2.3 above), and the *LoadFromDefinition* procedure that sets all properties specified in the definition file. Furthermore, any constructors and destructors present in the base class TBehavior should also be implemented and overridden (using the *override* directive). As a final step before using the new behavior (and in order for the UFLibrary to find the new class) the class must be registered in the run-time environment.

In Fig. 3, an actual class interface is shown. For the sake of both clarity and brevity, the interface shown corresponds to a trivial turning behavior. When active, this *turning* behavior will cause the robot to change its direction of heading by setting the motor output to two different values (`fLeftMotorOutput` and `fRightMotorOutput`), as seen in the implementation of the behavior's *Step* procedure in Fig. 4. Note that, in this very simple behavior, it would have been possible also to set the motor output already in the *Enter* procedure of the behavior.

3.2.2. *Writing a definition file*

The definition of the specific properties (such as parameter values and variable usage) of all behaviors is done in the robot definition file within the brain's list of behaviors (TBehaviorList). As an illustration, a simplified brain with only one behavior can be defined as shown in Fig. 5, where a simple exploration behavior is used as an example. In this behavior, the robot would normally move with constant motor torque, but would occasionally change direction (randomly) based on the readings of a laser range finder, using a slightly lower motor torque.

The behavior has one input variable (used internally by the behavior, in this case as a trigger for direction changes) and one state variable (used in the utility function), in this case the average reading of a laser range finder. Note that in this

```
TSimpleTurningBehavior = class(TMotorBehavior)
private
  fLeftMotorOutput: real;
  fRightMotorOutput: real;
public
  constructor Create; override;
  constructor CreateAndSet(B: TBehavior); override;
  function Copy: TBehavior; override;
  procedure LoadFromDefinition(ObjDef: TObjectDefinition); override;
  procedure Step(TimeStep: real); override;
  procedure Enter; override;
  procedure Exit; override;
  destructor Destroy; override;
end;
```

Fig. 3. Interface of a simple *turning* behavior.

```
procedure TSimpleTurningBehavior.Step(TimeStep: real);
begin
  fOutputVariables[1] := fLeftMotorOutput;
  fOutputVariables[2] := fRightMotorOutput;
end;
```

Fig. 4. Implementation of the *Step* procedure in the simple *turning* behavior.

simple case, with a single behavior, the actual values of the utility function would be of no use, since there is only one behavior to select.

As seen in this simple example, all objects are defined within `object...end` blocks. Immediately to the left of the word `object`, the name and the class of the object must be specified (separated by a colon). Everything contained within such a block consists of *properties* (such as e.g. the `ExplorationMotorOutput` property in Fig. 5) or further *objects*.

3.3. *Basic usage*

The use of the UFLibrary for the generation of a behavioral selection system for a robotic brain involves several steps, which can be summarized as follows (see also Fig. 6):

(1) Implement all behaviors that are to be included in the behavioral repertoire, unless the behaviors already provided in the UFLibrary are sufficient.
(2) Construct the body and brain of the robot by writing the corresponding definition file(s), specifying e.g. the state variables and input variables for each behavior, as well as the parameters of the behavior.
(3) Write the arena definition file by composing a list of available arena objects

```
Object Brain: TBrain
  Object Behaviors: TBehaviorList
    Level = 1

    Object Exploration: TExplorationBehavior
      ExplorationMotorOutput = 5.0
      TurnMotorOutput = 3.0

      Object InputVariables: TinputVariables
        Object InputVariable1: TsensorVariable
          CorrespondingSensorName= 'RangeFinder1'
        end
      end

      Object StateVariables: TStateVariables
        Object StateVariable1: TExternalVariable
          CorrespondingSensorName = 'RangeFinder1'
          ReadingProcedure = 'rpAverage'
        end
      end
    end #Exploration
  end #Behaviors
end
```

Fig. 5. A simple definition file for a robotic brain containing a single behavior (see Sect. 3.2.2).

(such as walls, doorways, etc.).

(4) Specify the ansatz for each utility function by providing the polynomial degree. (The number of variables in each utility function, equal to the number of state variables used in the behavior in question, is set automatically.)
(5) Specify the variation of the hormone functions for each behavior.
(6) Specify a fitness function, for example the distance traveled by the robot.
(7) Specify the termination criterion (or criteria) for the evaluation of an individual.
(8) Write the actual application, linking the UFLibrary to the executable file. This step can be made *very* simple, and amounts essentially to writing a simple GUI for passing information to and from the UFLibrary (something which is done very rapidly using Delphi). The source file of a basic, illustrative application is appended to UFLibrary, and can be used as a template, in order to further simplify this step.

At a first glance, the steps in the list above may seem quite complicated. However, as described in Sect. 3.2.1 above, step 1 in the list, i.e. the implementation of a behavior class derived from the base class TBehavior, is rather straightforward in most cases. For steps 2 and 3, pre-defined template files, such as those available in Ref. 17, can often be used. Step 4 amounts simply to providing an integer. Experience with the UF method shows that a value of around 2-4 (higher values

yielding somewhat better results) is appropriate [7], making this step almost trivial. Step 5 is currently needed, as explained in Sect. 2.2, but will be eliminated once the evolution of hormone functions has been included in the UFLibrary. Step 6 is generally simple, at least in situations involving a single task behavior. Step 7 is optional. By default, the simulation ends when the maximum simulation time has been reached. Since the UFLibrary relies upon the open-source software GLScene[19] for visualization, step 8 may require (if visualization is used) that this software library should be downloaded and installed.

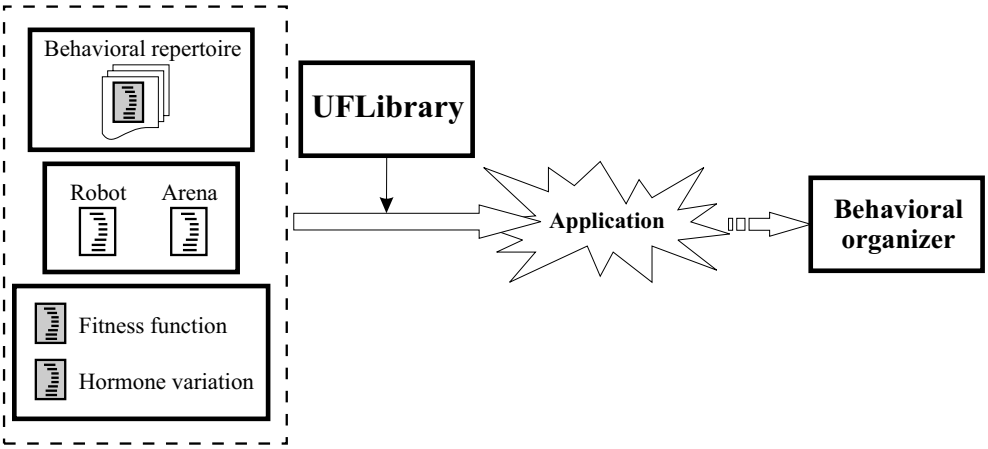


Fig. 6. Basic usage of the UF library. The dashed box indicates entities defined by the user. Within this box, white icons indicate text files, whereas gray icons correspond to source code.

4. Example: An Exploration Robot

The UFLibrary will now be illustrated by means of an example, namely a simple exploration robot, with a behavioral repertoire containing five behaviors. The actual program, i.e. the executable file (UFLibDemo), used for this example, as well as the definition files and additional relevant information can be found in Ref. 17.

4.1. *Description*

The task of the exploration robot is to navigate in an arena while at the same time avoiding collisions with (stationary) obstacles, and avoiding complete discharging of the battery. Such a robot could be used e.g. as a night watchman for an industrial facility. The robot used in this example is a differentially steered robot equipped with three sensors; (1) an internal battery sensor that measures the level of the battery, (2) a laser range finder measuring the distance to objects in a given sector, and (3) an IR detector for locating charging stations (represented by IR beacons).

In addition, the robot is equipped with two DC motors and a battery with a speed-dependent discharge rate defined as

$$\frac{dE}{dt} = -k_r - k_m\, v \tag{4}$$

where k_r and k_m are constants and v is the absolute speed of the robot. The constants in Eq. (4) where set so that the battery energy lasts around 60 s. during normal operation. Charging the battery takes a minimum of 10 s. Note that rather high discharge and charge rates were used in order to speed up the simulation. With these rates, the robot can go through several cycles of discharging and charging in a relatively short time, thus testing the behavioral selection system.

The behaviors included in the repertoire for the robotic brain are *Explore* (B1), *Avoid obstacles* (B2), and *Maintain energy* (B3) (all on the same hierarchical level), as well as *Locate charging station* (B3.1) and *Carry out charging* (B3.2), which are placed under B3, as illustrated in Fig. 7. The state variables were s_1, the sector

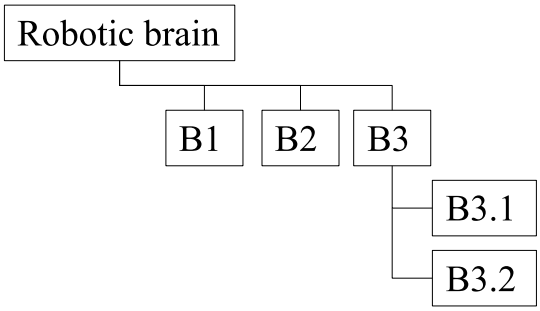


Fig. 7. Behaviors and hierarchy used in the exploration robot example. See the main text for a description of the behaviors.

average of distances measured by the laser range finder; p_1, the battery level; s_2, the reading of the IR detector; and x_1, x_2, and x_3, three hormone variables called *fear*, *hunger*, and *inverse satiation*, respectively. The roles of the hormone variables are approximately those indicated by their names. The utility function polynomials were of degree 2, and the state variables were used in the utility functions as follows: $U_1 = U_1(s_1, p_1)$, $U_2 = U_2(s_1, p_1, x_1)$, $U_3 = U_3(s_1, p_1, x_2)$, $U_{3.1} = U_{3.1}(s_2, p_1)$, $U_{3.2} = U_{3.2}(x_3)$. As mentioned above, the hormone variables, and the corresponding state variables, are here used in a simplified fashion: the variable x_1 is simply set to 1 when B2 is active, and to 0 otherwise (the setting of the hormone variable is handled by the entry procedure in B2). Similarly x_2 is set to 1 when any behavior under B3 is active, and to 0 otherwise. x_3 is, in the same way, set to 1 when B3.2 is active, and to 0 otherwise.

The behaviors were implemented in a rather simple fashion, in keeping with the UF method philosophy of delegating the difficult parts, such as the proper activation of behaviors, to the evolving behavioral selection system. B1 makes the robot travel

in straight lines and changing the direction of heading (in a deterministic fashion) at regular time intervals. B2 produces motor output such that the robot turns away from any nearby objects. If the utility of B3 exceeds that of B2 and B1, either B3.1 or B3.2 is selected for activation; B3 itself does not perform any actions other than selecting a behavior from its own list of behaviors. In B3.1, the robot starts to turn and if an IR beacon is discovered (by the IR detector), the robot travels to that IR beacon in a straight line. It should be noted that only B2 is capable of avoiding obstacles. Consequently, if an obstacle appears in front of the robot while B3.1 is active, the behavioral organizer must select B2 in order to avoid the obstacle, and then re-activate B3.1. Finally, B3.2 makes the robot remain in its position, so that *if* the robot is located at a charging station, it will charge its batteries.

Since the task of the robot is to explore the environment (see Fig. 8) as much as possible, B1 is the task behavior and the fitness increase for each activation of B1 is set equal to $\max(0, t_1 - 1)$, where t_1 is the time spent in B1, i.e. a fitness increment is only given when the robot spends at least one second in B1 (of course, other fitness measures would be possible as well, e.g. the distance traveled by the robot during an evaluation).

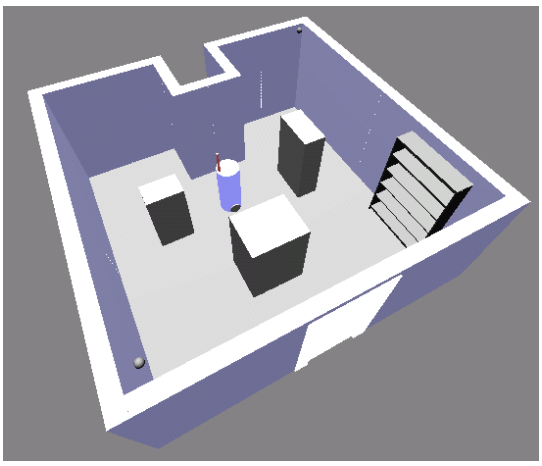


Fig. 8. Screenshot from the program UFLibDemo.exe used in the demonstration of the UFLibrary. The arena consists of a floor (5×5 meters) with surrounding walls. Inside the room there are four obstacles, two charging stations (not shown), and two IR beacons, shown as ball-like objects.

4.2. *Results*

Several runs were performed with the UFLibDemo program in order to find a behavioral organizer capable of selecting behaviors in such a way that the robot would manage to explore the arena while avoiding obstacles as well as complete discharging of its batteries.

Note that no noise was added in the runs presented here. However, the UFLibrary supports noise addition on all levels, e.g. in sensors and actuators. Clearly, any real robot will be subject to noise, both in its sensors and its actuators, and an accurate assessment of the performance of a simulated robot's brain, will generally require several separate evaluations, from which a fitness measure can be generated (e.g. as the average of the performance in each evaluation). However, in order to reduce running times (by keeping the simulations fully deterministic), noise was omitted in the particular example discussed here.

Typical parameters used during the runs are shown in Tab. 1. On a 3.2 GHz computer with 1 GB RAM, the UFLibDemo program evaluates around 5,000 individuals per hour. Apart from the different parameters for the EA shown in the

Table 1. Typical parameter settings used in the runs.

Time step length	0.01 s
Range of the maximum simulation time	$100 - 300$ s
Range of the polynomial degree of the utility functions	$2 - 4$
Range of the polynomial coefficients	$[-3, 3]$
Population size	100
Range of the crossover probability	$0.1 - 0.6$
Range of the parametric mutation rate	$0.03 - 0.05$
Tournament size	5
Tournament selection probability	0.7

table, various settings for e.g. the initial battery level and the starting position of robot were also tested.

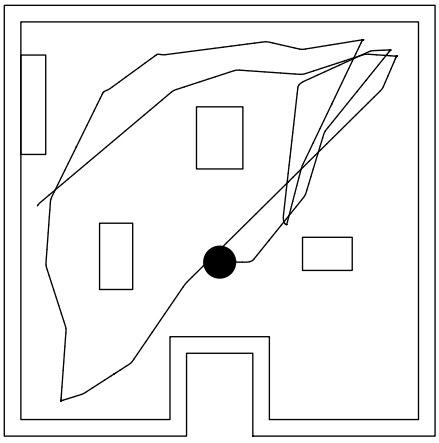


Fig. 9. Path traveled by one of the best robots (arena seen from above). The filled circle marks the robot's starting position. The robot first moves to one of the corners to recharge its batteries, and then proceeds to explore the arena, occasionally interrupting this activity to recharge its batteries, using either of the two charging stations.

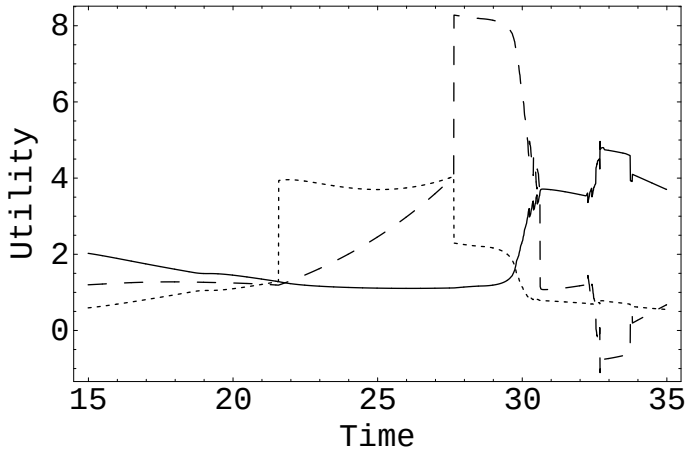


Fig. 10. Variation of utility functions associated with the behaviors B1 (*exploration*, solid line), B2 (*obstacle avoidance*, dashed), and B3 (*energy maintenance*, dotted) during a typical charging episode.

Due to the stochastic nature of the EA, the exact results varied from run to run. However, in general, the UFLibDemo program was able quickly to find appropriate utility functions, enabling the robot to perform its exploration task. In typical runs, evaluation of around 10^5 individuals was needed in order to arrive at a satisfactory result, implying run times of around one day.

In one of the best runs obtained, the polynomial degree of the utility functions was set to three, and the initial battery level to 50% of its maximum value. The robot was able to explore large parts of the arena (see Fig. 9) while avoiding battery depletion. In the particular run shown in Fig. 9, the robot utilized the charging stations (located in the upper right and lower left corners of the arena) several times in order to recharge its batteries and managed to explore the arena for a total of 198 seconds, before finally colliding with a wall.

The utility functions associated with B1, B2, and B3 during one of the charging sequences are illustrated in Fig. 10, where behavior B3 (*energy maintenance*) is active from around 21.5 s to 27.5 s. The robot then turns away from the wall by activating B2 (*obstacle avoidance*) for around 3 s, before activating B1 (*exploration*) again.

In other runs, the initial battery level was set relatively small (15% of the total capacity), requiring the robot quickly to find a charging station. Indeed, in these runs (one of which is illustrated in the form of a movie in Ref. 17), the evolved robotic brain quickly activates B3.1 and proceeds to one of the charging stations. While positioned near the charging station, the robotic brain activates B3.2, and the battery is recharged. Once the battery level is sufficiently high, the robotic brain activates B1 and starts exploring the room while avoiding any obstacles that may appear in the path of the robot by, in such cases, activating B2.

As a further illustration of the dynamics of behavioral selection, Fig. 11, shows the variation of U_1 and U_2 in a common situation were B2 is temporarily activated in order to avoid an obstacle. Note that the activation of B2 results from the combined effects of a reduction in U_1 and an increase in U_2.

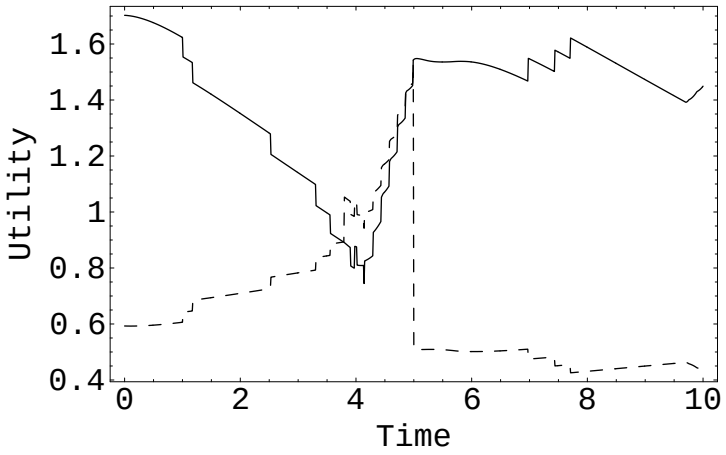


Fig. 11. Variation of the utility of B1 (solid) and B2 (dashed) when an obstacle is encountered. Initially, before the obstacle poses any threat, B1 has the highest utility and is therefore active. At around 3.5 s, a switch to B2 occurs as the utility of B2 increases when an obstacle is detected. When robot has turned so that it no longer senses the obstacle, the utility of B2 decreases rapidly (at 5 s) and the robot continues exploring the arena by activating B1 again.

5. Discussion and conclusion

In this paper, the UFLibrary software package has been introduced and illustrated by means of an example involving a simple exploration robot. It has been shown that the UFLibrary allows a user to evolve a behavioral selection system for a complex robotic brain containing several simple behaviors, with a minimum of effort. For example, given access to the UFLibrary and the template definitions files for the body and brain of the robot, and for the arena, the application (i.e. the executable file) and the definition files can be assembled in a matter of a few hours. Writing additional behaviors (derived from TBehavior) is also easy, the writing of a typical behavior taking around an hour or so.

Thus, when using the UFLibrary, the user must, as in *any* method for behavioral organization, supply *some* basic information, such as the individual behaviors as well as setup files, as explained in Sect. 3.3. However, it is important to realize that, with the exception of the hormone variables, a user of the UFLibrary is *not* required to determine, by hand, the criteria for appropriate activation of behaviors, a fact that distinguishes the UF method from most other methods for behavioral organization, as mentioned in Sect. 1.1.

The UFLibrary is provided as a software library rather than a fixed executable program. The reason for this approach is that, at least in the authors' experience, it is better to provide a library that gives the user the freedom to generate whatever executable file is needed for the project at hand, rather than attempting to generate a program that would be applicable in all possible investigations. All investigations differ in some respects and, with UFLibrary the user is not forced to use a pre-defined program that would perhaps, not be perfectly suited to the problem at hand. The drawback, of course, is that the user who wishes to generate a new executable program, must do some programming (and must have access to the Delphi programming environment[18] and GLScene[19]). However, such an executable file can be generated quite easily, using the sample source files appended to UFLibrary.

The example described in Sect. 4 illustrates the philosophy behind the UF method, which is to use simple individual behaviors, thus relegating the generation of complex overall behavior to the behavioral organizer. The rationale behind this approach is that the user must write the individual behaviors which, therefore, should be made as simple as possible, whereas the UF method should take care of the activation of behaviors at appropriate times. An additional advantage with this approach is that the evolutionary procedure, as implemented by the UF method, for generating a behavioral organizer based on utility functions (and, eventually, hormone functions), can be specified once and for all, as is done in the UFLibrary described in this paper.

Note also that the user has a great deal of freedom when writing specific implementations of a given behavior. As long as the obvious requirements are fulfilled, e.g. that the behavior should only make use of readings from sensors that are actually available on the robot, vastly different implementations of the individual behaviors can be used, without any modification of the UF method, i.e. the procedure for generating the behavioral *organizer*.

A possible drawback of the UF method is that the heavy use of an EA requires that the generation of the behavioral organizer must be carried out in simulation, evolution directly in hardware being too time-consuming and difficult to monitor. Of course, no simulation can capture accurately all aspects of reality, meaning that some fine-tuning and, possibly, iterations involving repeated simulations and hardware tests, must be performed. However, this is a rather small price to pay for the ability to construct a general behavioral selection system of arbitrary complexity, using a minimum of hand-coding. The issue of transferring the results obtained with the UFLibrary to actual robots is a topic for current and future research, the results of which will be presented elsewhere.

Finally, it should be noted that the UF method optimizes the selection of behaviors within a hierarchy defined by the user, as shown e.g. in Figs. 2 and 7. In principle, the method could be extended to allow the EA automatically to generate the actual structure of the brain (which is currently supplied as a definition file, based e.g. on the template provided in Ref. 17). In the authors' experience, providing the structure of the robotic brain is often quite straightforward, so that the

need for making also this step automatic is quite limited. However, the issue may become relevant for very complex robotic brains.

Another important extension to the current version of the UFLibrary will be to allow a more general specification of the hormone variables, by *evolving* their variation (e.g. as polynomials, with the state variables as arguments) rather than specifying it by hand as in the current version, thus completely eliminating the need for hand-coding in the specification of the criteria for behavioral selection.

Acknowledgments

The authors would like to thank the Carl Trygger foundation for financial support for this project.

References

1. R.C. Arkin, *Behavior-based robotics*, (MIT Press, 1998)
2. S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd Ed., (Prentice Hall, 2002)
3. P. Pirjanian, *Behavior coordination mechanisms – state-of-the-art*, Technical Report IRIS-99-375, Institute of Robotics and Intelligent Systems, USC, Los Angeles, (1999)
4. M. Wahde, *A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions*, J. Systems and Control Engineering, 217, pp. 249–258, (2003)
5. B.M. Blumberg, *Action selection in Hamsterdam: Lessons from ethology*, In: From Animals to animats 3, Proc. of the 3rd Int. conf. on simulation of adaptive behavior (SAB94), (MIT Press, 1994)
6. P. Maes, *Modeling adaptive autonomous agents*, Artificial Life **1**, pp. 135–162, (1994)
7. M. Wahde, J. Pettersson, H. Sandholt and K. Wolff *Behavioral Selection Using the Utility Function Method: A Case Study Involving a Simple Guard Robot*, to appear in Proc. of the 3rd Int. Symp. on Autonomous Minirobots for Research and Edutainment (AMIRE 2005), (2005)
8. J. Pettersson and M. Wahde *Application of the utility function method for behavioral organization in a locomotion task*, IEEE Trans. Evol. Comp., (2005, in press)
9. D. McFarland, *Animal Behavior*, 3rd edition, (Addison-Wesley, Harlow, 1999)
10. D. McFarland and T. Bösser, *Intelligent Behavior in Animals and Robots*, (MIT Press, 1993)
11. D. McFarland and E. Spier, *Basic cycles, utility, and opportunism in self-sufficient robots*, Robotics and Autonomous Systems, **20**, pp. 179–190, (1997)
12. O. Michel/Cyberbotics Ltd, *Webots™: Professional Mobile Robot Simulation*, pp.39–42, International Journal of Advanced Robotic Systems, Volume 1 Number 1, (2004)
13. N. Koenig and A. Howard, *Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator*, IEEE/RSJ International Conf. on Intelligent Robots and Systems, (Sep. 2004)
14. C. Leger, *Darwin2K: An Evolutionary Approach To Automated Design For Robotics*, (Kluwer Academic Publishers, 2000)
15. S. Nolfi, *EvoRobot 1.1 User Manual*, Institute of Psychology, Rome, Italy, (2000)
16. F. Delcomyn and J.A. Reichler, *Dynamics Simulation and Controller Interfacing for Legged Robots*, The International Journal of Robotics Research, Vol. 19, No. 1, pp. 41–57, (Jan. 2000)

17. UFLibrary demo, http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary/Demo.html
18. Borland Delphi, http://www.borland.com/delphi/
19. GLScene, http://glscene.org/

Paper VI

Behavioral Selection using the Utility Function Method: A Case Study Involving a Simple Guard Robot

in

Proceedings of the 3rd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005), Fukui, Japan, September 2006, pp. 261–266.

Behavioral Selection Using the Utility Function Method: A Case Study Involving a Simple Guard Robot

Mattias Wahde[1], Jimmy Pettersson[1], Hans Sandholt[1], and Krister Wolff[1,2]

[1] Department of Applied Mechanics, Chalmers University of Technology, 412 96 Göteborg, Sweden
{mattias.wahde, hans.sandholt, jimmy.pettersson}@chalmers.se
[2] Department of Microtechnology and Nanoscience, Chalmers University of Technology, 412 96 Göteborg, Sweden
krister.wolff@mc2.chalmers.se

Summary. In this paper, the performance of the utility function method for behavioral organization is investigated in the framework of a simple guard robot. In order to achieve the best possible results, it was found that high-order polynomials should be used for the utility functions, even though the use of such polynomials, involving many terms, increases the running time needed for the evolutionary algorithm to find good solutions.

1 Introduction

In behavior-based robotics (BBR) [1], the artificial brain of a robot is built in a bottom-up fashion, starting from simple low-level behaviors. An obstacle facing the behavior-based approach is the problem of behavioral selection, i.e. the problem of activating appropriate behaviors at all times. In simple robots, with small behavioral repertoires, the selection of behaviors (for activation) can be generated manually, which indeed is what is done in most methods for behavioral selection [4, 5, 6].

However, in robots with larger behavioral repertoires, specifying behavioral selection by hand is a daunting task, not least because of the difficulty in comparing, at all times and in all situations, the relative merits of several behaviors. Such comparison requries a common currency which, in economic theory and game theory, goes under the name *utility*, a concept that has also been introduced in ethology and, more recently, in robotics [3, 4].

In order to overcome the difficulties associated with behavioral selection, a method known as the *utility function* (UF) *method* has been developed [4]. In this method, behavioral selection is based on the value of utility functions

that are *evolved* rather than hand-coded, thus minimizing the bias introduced by the user of the method.

In this paper, the UF method will be illustrated by means of an example, namely a simple simulated guard robot. In addition, the performance for various utility function specifications will be studied.

2 The Utility Function Method

Due to space limitations, only a brief description of the UF method will be given here. A more complete discussion of the method is available in [4]. In the UF method, each behavior $B_j, j = 1, \ldots, N$, where N is the number of behaviors, is associated with a utility function, whose variables are (a subset of) the state variables of the robot. The state variables are of three kinds: External variables, denoted s_i (e.g. the readings of IR sensors on the robot), internal physical variables, denoted p_i (e.g the readings of a battery sensor), and internal abstract variables, denoted x_i. The latter correspond to the readings of internal variables known as hormones in the UF method. In general, each utility function is given by a polynomial ansatz. For example, the ansatz for a second-degree polynomial utility function of two variables s_1 and x_1 is given by

$$U(s_1, x_1) = a_{00} + a_{10}s_1 + a_{01}x_1 + a_{20}s_1^2 + a_{11}s_1x_1 + a_{02}x_1^2. \tag{1}$$

The UF method is an arbitration method, i.e. a method in which one and only one behavior is active at any given time. Behavioral selection is simple in the UF method: at all times, the behavior whose utility function takes the highest value is activated. The problem, of course, is to specify the utility functions so as to generate purposeful and reliable behavioral selection. In the UF method, this is done using an evolutionary algorithm (EA). As in any EA, a fitness function must be specified. In the UF method, the fitness is often associated with the execution of a given task behavior, the other behaviors being considered as auxiliary behaviors, i.e. behaviors which are needed (such as battery charging), but which do not increase the fitness of the robot. Once the fitness function has been specified, the task of the EA is thus to set the coefficients of the N polynomial utility functions.

An interesting question in this regard concerns the number of such coefficients, which, in turn, determines the complexity of the problem that the EA must solve. In general, it can be shown that a polynomial function of n variables and of degree p contains

$$\binom{n+p}{p} = \binom{n+p}{n} \tag{2}$$

distinct terms. (For example, in Eq. (1), $n = 2$ and $p = 2$, so that the number of terms, according to Eq. (2), equals $\binom{4}{2} = 6$).

3 Case Study: A Simple Guard Robot

As a case study, consider a simple simulated guard robot whose task it is to patrol the arena shown in the left panel of Fig. 1. The arena contains numerous obstacles in the form of pillars, as well as three battery charging stations, located at corners in the arena. The simulated robot is a differentially steered, two-wheeled robot, with two DC motors. The robot will be equipped

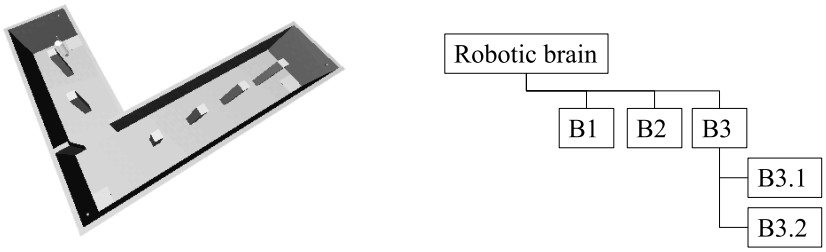


Fig. 1. Left panel: The arena patrolled by the guard robot. Right panel: Behavioral hierarchy of the robotic brain.

with five behaviors, namely *straight-line navigation* (B_1), *obstacle avoidance* (B_2), *energy maintenance* (B_3), *corner seeking* ($B_{3.1}$), and *battery charging* ($B_{3.2}$). In the UF method, as implemented in the UFLibrary software library currently under development at Chalmers University of Technology, behaviors are organized in hierarchies, as indicated in the right panel of Fig. 1. Utility functions are compared on a level-by-level basis. Thus, for each time step, it is determined which of the three functions U_1, U_2, and U_3 takes the highest value. If it happens to be U_3, the comparison of $U_{3.1}$ and $U_{3.2}$ will determine which of these two behaviors is active.

In B_1, the robot simply moves in a straight line, by setting its motor outputs to equal values. In B_2 the robot turns until no obstacle is visible in front of it, and then stops. If $B_{3.1}$ is active, the robot will rotate in an attempt to find a charging station (each of which is associated with an IR beacon detectable by a sensor on the robot). In $B_{3.2}$ the robot remains at a standstill, charging the batteries *if* it happens to be at a charging station.

Since the task of the robot is to cover as much of the arena as possible, a suitable fitness measure is simply the time spent in the navigation behavior B_1. In order to avoid rapid swapping between behaviors, a slightly modified fitness function was used, however, in which the robot only obtains a fitness increase if it spends at least one full second executing B_1.

Due to space limitations, the full ansatz for each utility function will not be given here. Suffice it to say that the number of variables in $U_1, U_2, U_3, U_{3.1}$, and $U_{3.2}$ were 4, 3, 3, 2, and 1, respectively. Thus, for degree p, the number of polynomial coefficients that must be determined by the EA equals

$$n_c = \binom{4+p}{4} + 2\binom{3+p}{3} + \binom{2+p}{2} + \binom{1+p}{1}. \quad (3)$$

4 Simulation program

A simulation program was written in Delphi object-oriented Pascal [2] using the UFLibrary software package. The UFLibrary provides a general implementation of the UF method, handling all issues involving the evolution of polynomial utility functions for behavioral selection. The task of the user is to provide (1) the constituent behaviors for the behavioral repertoire, (2) the polynomial degree p of the utility functions, (3) a fitness function, (4) the arena, in the form of a text file readable by the UFLibrary, (5) the definition of the robot body and the general structure of its brain (as shown in the right panel of Fig. 1), also in the form of a text file in a given format. The specification of the robotic brain also involves specifying the state variables included in each utility function polynomial.

5 Results

For simplicity, the simulations were performed without any sources of noise. Each evaluated individual was allowed a maximum simulation time of $100s$. However, the evaluation of a simulated robot was terminated directly in case of collisions with obstacles or if the on-board battery became fully discharged. In each run, 10,000 individuals were normally evaluated, even though some extended runs were carried out as well (see below).

Four different polynomial degrees were investigated, namely $p = 1, 2, 3$, and 4, for which the number of polynomial coefficients (n_c) equals 18, 44, 89, and 160, respectively. In order to allow a fair comparison between the performance of the EA for different values of p, mutation rates (p_m) were set to $1/n_c$ or $3/n_c$. The population sizes n_p were set to 20 or 100 individuals. In the UFLibrary, the crossover procedure swaps entire polynomials between chromosomes. Here, the crossover probability p_c was set to 0.2 or 0.8. Furthermore, tournament selection was used, with the tournament size n_t set to 10% of the population size. Thus, a total of $4 \times 2 \times 2 \times 2 = 32$ parameter combinations were investigated. Furthermore, because of the stochasticity of EAs, several (N_R) runs had to be performed for each setting, in order to form a reliable average. Since a typical run lasted for a few hours, N_R was limited to 5-7, resulting in a total of around 200 runs.

Due to the richer dynamical structure accessible in runs with large p, it could perhaps be expected that these runs would outperform those with lower p values. However, no such simple trend was found: The averages (over all runs with a given value of p) of the best fitness values found after 10,000 evaluated individuals, were found to be 10.36, 9.33, 10.31, and 9.43, for $p = 1, 2, 3$, and

Table 1. Averages of the best fitness values found after 10,000 individuals, for runs with different values of the polynomial degree p and the mutation rate p_{m}.

	Polynomial degree			
p_{m}	1	2	3	4
$1/n_c$	8.973	6.396	7.255	5.291
$3/n_c$	11.76	11.73	13.35	13.59

4, respectively. As can be seen in Table 1 there is, on the other hand, a strong trend in favor of the larger of the two mutation rates. A similar, albeit weaker, trend (not shown) was also found in favor of large population sizes, whereas no discernible trend was encountered for the crossover probability. Note also that, within the categories shown in Table 1, the spread between different runs was quite large, with the majority of runs reaching rather low fitness values.

6 Discussion and conclusion

While the average results obtained from the runs performed here show only very little variation with the polynomial degree p, this does not necessarily imply that the choice of p does not matter. Two possible interpretations of the results are that (1) perhaps larger values of p do indeed make it possible to achieve better results but that *finding* such solutions becomes progressively more difficult as p is increased (due to the large increase in the size of the search space), or (2) maybe $p = 1$ or 2 is sufficient for the problem at hand and that, in the runs with larger p, the coefficients in front of the third and fourth-order terms are simply eliminated by the EA. However, an inspection of those coefficients showed the latter *not* to be the case.

Table 2. Averages of the three best fitness values found for each polynomial degree p in the extended runs.

Polynomial degree			
1	2	3	4
23.24	37.23	47.91	49.33

Evidence in favor of the first interpretation can be found, on the other hand: As shown in Table 1, the increase in performance as the mutation rate is raised is stronger for large p values than for small ones, indicating that the larger p values require a more thorough inspection of the search space before the best solutions can be found. In order to test this tentative conclusion further, several extended runs were performed, the results of which are summarized in Table 2. Note that the extended runs differed somewhat in length: The number of evaluated individuals was on the order of 50,000

to 100,000. For this reason, the table shows an average of the three best results obtained for each p. In Table 2, the results clearly show an increase in performance as p is raised. Thus, it may be concluded that the choice of p strongly influences the quality of the results that can be achieved, but that the benefits of larger p values only become evident after the evaluation of a large number of individuals.

Finally, note that the chosen fitness measure (time spent in behavior B_1) does not lead to an incentive for the robot to explore the whole area. However, such solutions were indeed found by the EA in some runs. One example is shown in Fig. 2, with a fitness value equal to 44.57 found after the evaluation of 41,400 individuals.

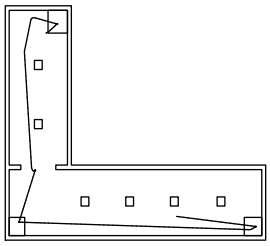


Fig. 2. The motion of one of the best robots found by the EA. The squares in the upper right, lower left, and lower right corners represent the charging stations.

Acknowledgments

The authors wish to thank the Carl Trygger foundation for financial support for this project.

References

1. Arkin, R.C., Behavior-based robotics, MIT Press, 1998
2. `http://www.borland.com/delphi`
3. McFarland, D. and Bösser, T. Intelligent behavior in animals and robots, MIT Press, 1993
4. Wahde, M., A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions, Journal of Systems and Control Engineering, 217, pp. 249–258, 2003
5. Maes, P., How to do the right thing, Journal of Connection Science, 1, No.3, pp. 291–323, 1989
6. Blumberg, B.M., Action selection in Hamsterdam: Lessons from ethology, In: From Animals to animats 3, Proc. of the 3rd Int. conf. on simulation of adaptive behavior (SAB94), MIT Press, 1994

Paper VII

Behavioral selection in domestic assistance robots: A comparison of different methods for optimization of utility functions

to appear in

Proceedings of the 2006 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2006), Taipei, Taiwan, October 2006.

Behavioral selection in domestic assistance robots: A comparison of different methods for optimization of utility functions

Jimmy Pettersson, David Sandberg, Krister Wolff, and Mattias Wahde

***Abstract*— In this paper, the performance of several evolutionary algorithms (EAs), involving different operators, is investigated in connection with the utility function (UF) method, a method for generating behavioral organization (selection) systems in autonomous robots.**

The standard UF method, which uses an ordinary genetic algorithm (GA) with fixed-length chromosomes is compared with modified evolutionary methods in which the chromosomes are allowed to vary in size.

The results show that, contrary to expectations, the standard UF method performs at least as well as the modified methods, despite the fact that the latter have larger flexibility in exploring the space of possible utility functions. A tentative explanation of the results is given, by means of a simple, analytically tractable, behavioral organization problem.

I. INTRODUCTION AND MOTIVATION

As the percentage of elderly people increases, there will be an increase in the demand for domestic robots. These robots are expected to perform tasks such as notifying the owner about periodic events (e.g. intake of medicine), alerting in case of danger, providing transportation assistance, serving as an interface to nurses or doctors, vacuuming floors, and partly filling the need for social stimulation. Already today, there are numerous robots available, designed for domestic use, with vacuum cleaning and entertainment being the most common applications.

For domestic robots to become widely used, they need to become more intelligent and to be able to handle reliably a multitude of tasks. In behavior-based robotics (BBR) [1], the aim is to develop truly intelligent robots, capable of performing their duties in unstructured environments, through the combination of several low-level behaviors. In BBR, a robotic brain commonly consists of a repertoire of behaviors, and a system (the behavioral organizer) responsible for activating the correct behavior at any given time. Thus, a major issue in BBR is to determine *when* (i.e. under what circumstances) a behavior from the repertoire should be selected for activation. In order to solve this problem, several methods for behavioral organization (or behavior selection) have been proposed [2], [3].

However, most such methods require the user to specify numerous parameters by hand. One of the aims of the recently introduced utility function (UF) method [3] is to free the user from this burden. In the UF method, the relative merit (utility) of each behavior is obtained from a utility function (normally a polynomial). The utility functions determining the behavior selection in a given robotic brain are optimized by means of an evolutionary algorithm (EA).

With the UF method, selection among behaviors is simple, as soon as the utility functions have been obtained: At all times, the behavior associated with the highest utility value, as obtained from the utility function, is selected for activation.

In the UF method, each utility function depends on several state variables (see below), and the normal procedure is to use *complete* polynomials to represent the utility functions, i.e. polynomials containing all possible combinations of variables, up to a maximum degree d. For example, a complete utility polynomial $U_{\rm c}(s_1, s_2)$ of two variables and degree 2 would take the following form

$$\begin{aligned} U_{\rm c}(s_1, s_2) &= a_{00} + a_{10}s_1 + a_{01}s_2 + a_{20}s_1^2 + \\ &+ a_{11}s_1s_2 + a_{02}s_2^2, \end{aligned} \tag{1}$$

where the a_{ij} are constants to be determined by the EA. However, it is evident that the number of terms in such a polynomial representation grows rapidly with the number of variables n and the polynomial degree d, thus making the search space very large. In view of the long evaluation times (several seconds up to several minutes per individual) in many behavior selection problems, the need for efficient optimization methods (for the utility functions, in the case of the UF method) is evident.

An obvious alternative to using complete polynomials would be to use *non-complete* polynomials, i.e. polynomials lacking certain terms. A non-complete version ($U_{\rm nc}$) of $U_{\rm c}$ could, for example, take the form

$$U_{\rm nc}(s_1, s_2) = a_{00} + a_{20}s_1^2 + a_{11}s_1s_2, \tag{2}$$

or any other form involving a subset of the 6 terms in (1). Provided that the EA would be able (through crossover and mutations) to modify the exact nature of the terms in the non-complete polynomials, one might expect that such a representation would lead to faster optimization of the utility functions.

In this paper, the performance of different EAs will be studied for the case of a simple domestic exploration robot, which can serve as a metaphor for e.g. a vacuum cleaning robot. Specifically, the performance of a standard genetic algorithm (GA), which uses complete polynomials, is compared to the performance of several modified GAs, which use incomplete (or at least variable-structure) polynomials.

This work was supported by the Carl Trygger foundation.

The authors are affiliated with the Department of Applied Mechanics, Chalmers University of Technology, 412 96 Göteborg, Sweden. Corresponding author's e-mail: mattias.wahde@chalmers.se

II. THE UTILITY FUNCTION METHOD

In the UF method, each behavior in the behavioral repertoire $\{B_1, B_2, \ldots, B_N\}$ is associated with a utility function

$$U_i(\mathbf{s}, \mathbf{p}, \mathbf{x}), \quad i = 1, 2, \ldots, N, \tag{3}$$

where $\mathbf{s}$ is a set of external variables (such as external sensor readings), $\mathbf{p}$ is a set of internal physical variables (such as wheel encoder signals), and $\mathbf{x}$ is a set of internal abstract variables (corresponding to hormones in a biological organism).

The standard UF (hereafter: SUF) method sets up a population in which the chromosomes specify the coefficients of the utility functions for the individual in question. In the initial population, the coefficients are set to random values within a given range. Each individual is then evaluated, by running a simulation in a given arena while monitoring the performance (fitness) of the individual. The exact nature of the fitness measure will, of course, vary from problem to problem.

Being an arbitration method, i.e. one in which only a single behavior is allowed to be active at any given instant, the UF method selects, during the operation of a robot, the behavior whose utility function has the highest current value. The utility functions are optimized using an EA in which the fitness value only increases during the active period of specific behaviors, the *task* behaviors. Behaviors that do not modify the fitness value are referred to as *auxiliary* behaviors.

Once all individuals have been evaluated, new individuals are formed through the procedures of fitness-proportional selection, crossover, and mutation. The new generation thus formed is then evaluated in the same way as the first, etc. For a more detailed discussion of the UF method, see [3].

As mentioned above, in the SUF method, each utility function is represented by a complete polynomial, including all terms up to a pre-specified degree d. If the number of arguments, formed by the union of the sets $\mathbf{s}$, $\mathbf{p}$, and $\mathbf{x}$, is equal to n, it can be shown that the total number of terms in the polynomial is equal to

$$N_t = \binom{n+d}{d} = \binom{n+d}{n}. \tag{4}$$

An example with $n = 2$ and $d = 2$ is shown in (1). Denoting the full variable set $\{\mathbf{s}, \mathbf{p}, \mathbf{x}\}$ by $\mathbf{z}$, the utility functions can thus be written as sums of terms of the form

$$c z_1^{k_1} z_2^{k_1} \cdots z_n^{k_n}, \tag{5}$$

where c is a constant, and the k_i are non-negative integers. A utility function polynomial is said to consist of *unique terms* if any given exponent combination $(k_1, k_2, \ldots, k_n)$, is represented *at most* once in the polynomial. As indicated in connection with (1) above, a polynomial is said to be *complete* if all exponent combinations occur *exactly* once. Thus, in the case of the SUF method, the structure of the utility functions remains intact throughout the optimization process. However, using the methods defined in Sect. IV-B, structural modifications are introduced that enable a variable length of the polynomials, and thus the formation of non-complete polynomials.

III. SIMULATIONS

Simulations were made using software based on the UFLibrary software package [4]. Implemented in object-oriented Pascal, the UFLibrary provides a rapid way of generating behavior selection systems and provides a general framework for the UF method. In addition, the UFLibrary also contains methods for visualization, utilizing the OpenGL interface, methods for solving and integrating the dynamics associated with differentially steered robots, various sensor models etc.

A. Simulation setup

The arena used in the experiments is a small apartment having three rooms, as shown in Fig. 1. Specific areas of the apartment, for instance the bathroom, are considered to be off-limits, and have therefore been omitted from the model.

Cylindrical in shape, the simulated robot is differentially steered, powered by two DC motors, each modeled with a simple DC circuit. For sensing, the robot is equipped with five IR sensors placed symmetrically on the front of the robot in the directions $-60°$, $-30°$, $0°$, $30°$, and $60°$, respectively. Each sensor has an opening angle of 0.5 radians, a range of 0.5 m, and uses a model based on a ray-tracing technique as suggested in [5] for calculating the (diffuse) sensor reading.

In addition to the measurements provided by the IR sensors, the robot is capable of measuring the amount of energy available in its onboard battery. As the robot moves, the battery discharges as

$$\frac{\mathrm{d}E}{\mathrm{d}t} = -k_r - k_m|v| \ , \tag{6}$$

where k_r and k_m are positive constants and v is the speed (with sign) of the robot. The constants in (6) were set so that the battery would last approximately 25 s. Charging of the battery occurs with a constant rate (provided that the robot is currently executing the battery charging behavior, see below) as $\mathrm{d}E/\mathrm{d}t = k_c$, with k_c set so that an empty battery becomes fully charged after 10 s of continuous charging. In all runs, the initial energy level was set to half the battery's capacity.

In order for the robot to be able to explore the arena, three different behaviors were included in the behavioral repertoire, namely *straight-line navigation* (B1), *obstacle avoidance* (B2), and *battery charging* (B3). Each behavior is responsible for setting the command signals to the two DC motors in a certain way. If B1 is active, the motor command signals are set to positive values (equal in magnitude), causing the robot to travel forward in a straight line. In B2, the motor signals are set to be equal in magnitude but with opposite signs, making the robot turn on the spot, after an initial transient in case B1 was active before the activation of B2. The turning is aborted, and the motor signals are thus set to zero, when the sector in front of the robot is free from objects. When B3 is selected for activation, the motor signals are set to zero and the robot stops.

For simplicity, battery charging was modelled in a very simple way: In order to charge the battery, the robot simply has to activate B3, thus avoiding the need of additional behaviors for locating and approaching charging stations. Such behaviors would require additional sensors for identifying a charging station as well as methods for localization.

Since the task of the robot is to explore the environment, fitness is associated with the time spent in B1 (the task behavior). Every time the robot uses B1 it receives a fitness increase (upon leaving B1, or upon termination of the simulation) equal to $\max(0, t_1 - 1)$, where t_1 is the time spent in B1 since its latest activation. Thus, fitness is only increased if B1 is active for a period longer than one second. By incorporating this threshold in the fitness function, behavior dithering (or behavior mixing) is effectively eliminated. A situation involving behavior dithering could, for example, occur between behaviors B1 and B3. Due to the fact that battery charging occurs whenever B3 is active, without regard to the current position of the robot, the behavior selection could be optimized in such a way that B1 and B3 would be effectively mixed by toggling between B1 and B3 every time step. Due to the dynamical properties of the robot, this would enable the robot to travel forward at low speed while, at the same time, charging its onboard battery. This situation is clearly unrealistic and was eliminated by introducing the one-second threshold in the fitness function.

For the three behaviors B1, B2, and B3, the utility functions were specified as

$$\begin{aligned} U_1 &= U_1(s_1, s_2, s_3, s_4, s_5, p_1) \\ U_2 &= U_2(s_1, s_2, s_3, s_4, s_5, p_1, x_1) \\ U_3 &= U_3(p_1, x_2) \end{aligned} \tag{7}$$

where s_i are the readings of the five IR sensors, p_1 is the amount of energy stored in the onboard battery, x_1 is a hormone variable corresponding to *fear*, and x_2 is a hormone variable corresponding to *inverse satiation*. The total number of polynomial terms in a chromosome representing utility functions for the SUF method can easily be calculated, using (4), as

$$N_t = \binom{9}{3} + \binom{10}{3} + \binom{5}{3} = 214\,. \tag{8}$$

Collisions with objects in the environment, as well as battery depletion, cause the simulation to be aborted. Thus, in order for the robot to receive high fitness values, the auxiliary behaviors B2 and B3 must be activated every now and then for collision avoidance and battery charging, respectively.

Since the focus of this paper is to compare different evolutionary approaches to the generation of utility-function based behavioral organization systems, rather than the application *per se*, several simplifications have been made: As indicated above, the behaviors B1-B3 are very simple, and so is the arena. Furthermore, the exploration carried out by the robot does not include any measure of the covered area and neither does it depend on any map. In addition, the rates of battery charging and discharging have been set to very high values, in order for the robot to be forced to activate all three behaviors even during a rather short simulation. Finally, noise was omitted from the analysis, and all simulations were performed using the same initial setup (i.e. placement and orientation of the robot in the arena).

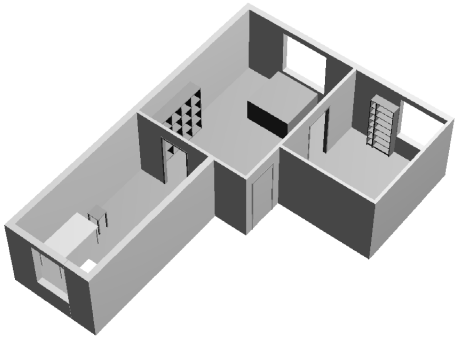


Fig. 1. Arena used in the simulations (view from above).

IV. METHODS

A. Standard UF method

The UFLibrary implements the SUF method, with a GA in which each gene in the chromosome encodes a complete polynomial, representing the utility function associated with a behavior. By default, a single-point crossover operator is used that swaps genes between two selected parents, as exemplified in the top panel in Fig. 2. Hence, the crossover operator effectively swaps entire polynomials, keeping the structure of each polynomial intact. The mutation operator used in this method modifies the coefficients of the polynomial terms, but not the exponents of the variables, again keeping the structure of the polynomials intact.

B. Modified methods

Several modified methods, involving evolutionary operators different from those used in the SUF, were tried, namely

- M_1 Initialization[1] by generating $N = N_t$ (see (4)) random polynomial terms up to a given maximum degree d. In M_1 the initial polynomials are complete, i.e. there exists one, and only one, term (and thus one coefficient) for each exponent combination $(k_1, \ldots, k_n)$, see (5).
- M_2 Initialization by generating polynomials with $N \in [1, N_t]$ random terms up to a given maximum degree d. Thus, in this method, the initial population will contain utility functions with fewer terms than in M_1. Furthermore, in M_2, uniqueness is not enforced even in the initial population. Thus, the polynomials *may* contain more than one term for a given exponent combination $(k_1, \ldots, k_n)$ (again, see (5)).
- M_3 As M_1, but with the additional operation of simplifying the polynomials resulting from the crossover operation to ensure that the polynomials consist of unique terms as defined in Sect. II. Thus, if, for example, an offspring chromosome contains the two terms $c_1 z_1^{k_1} \cdots z_2^{k_n}$

[1] i.e. generation of the initial population.

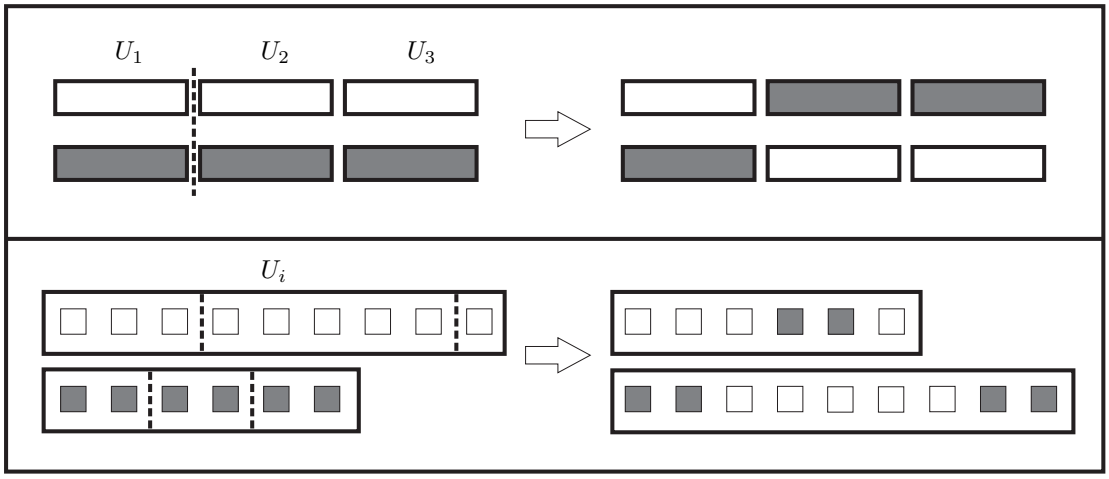


Fig. 2. Crossover operators used in the standard UF (SUF) method (top panel) and in the modified methods (bottom panel). Genetic material associated with the two parent chromosomes is indicated with the colors white (first parent) and gray (second parent), and dashed lines indicate the randomly chosen crossover points. In the top panel, the crossover operator effectively swaps entire polynomials between the two parents as the genes (three in this case) encode the polynomials associated with each behavior. In the modified methods, the crossover operator swaps segments consisting of polynomial terms, which are illustrated as small boxes in in the bottom panel. Note that, for the modified methods, even if the parent chromosomes contain unique polynomial terms, the offspring may contain non-unique terms as a result of crossover.

and $c_2 z_1^{k_1} \cdots z_2^{k_n}$ (with the same exponent combination $(k_1, \ldots, k_n)$), they are fused into *one* term with coefficient $c = c_1 + c_2$.

M_4 As M_2, but with the same simplification procedure as in M_3.

M_5 As M_1, but with an additional mutation operator that randomly selects one existing term and generates a new coefficient *and* new exponent values (with a maximum degree d). Hence, c, k_1, k_2,..., k_n in (5), for the chosen polynomial term, are assigned new random values.

M_6 As M_2, but with the same additional operator as in M_5.

In the methods M_1–M_6, a two-point crossover operator was used, as illustrated in the bottom part of Fig. 2. Contrary to the crossover operator used in the SUF method, in which entire polynomials are swapped, the two-point crossover operates on the genetic material contained *in* the polynomials, i.e. on the level of polynomial terms. For each polynomial in the chromosome, two segments of random lengths are swapped between the two participating individuals. This operator effectively swaps sequences of polynomial terms, changing the structure of the involved polynomials.

Mutation was used in the same way as in the SUF method, i.e. randomly changing the coefficients (c) of the polynomial terms, leaving the combination of exponents unchanged, except in M_5 and M_6, where exponents could be modified as well (see above).

In both the SUF method and in the modified methods (M_1–M_6), generational replacement and elitism were used in the implementation of the algorithms. Thus, when forming the new generation through selection, crossover, and mutation, the best individual in the last generation was transferred unchanged into the new generation.

V. RESULTS

In all runs, a population size of 100 individuals was used and the EAs were normally run for 50 generations (5 000 evaluated individuals). The maximum simulation time was set to 100 s, exceeding the time it takes for a fully charged battery to deplete. Selection was performed using tournament selection with a tournament size of 5 and a probability of selecting the best individual set to 0.7. The probability of crossover was set to 0.5.

In the runs involving the SUF method, a fixed polynomial degree of 3 was used, as suggested by the results in [6]. For the runs made with the modified methods, the same degree was used as an upper bound of the generated polynomial terms.

In the SUF method, a mutation rate of $3/N_t$ was chosen, based on the results reported in [6]. Note that N_t is the number of polynomial terms in a complete polynomial, as defined in (4).

In the methods M_1–M_6, a variable mutation rate of $1/n_c$ was used[2], where n_c denotes the (variable) total number of terms in the utility functions of a given individual. An exception was the exponent mutations in M_5 and M_6, which occurred with the rate $1/(5n_c)$.

Due to the stochastic nature of EAs, several runs should be made in order to evaluate the performance of any given method. Here, 10 runs were made for each method, and the average $\bar{f}$ of the best fitness f^{best} obtained in each run was formed. Thus, for any given method,

$$\bar{f} = \frac{1}{10} \sum_{i=1}^{10} f_i^{\text{best}}. \tag{9}$$

[2]Some test runs were carried out using a mutation rate of $3/n_c$ (for M_1–M_6), but these runs did not give better results.

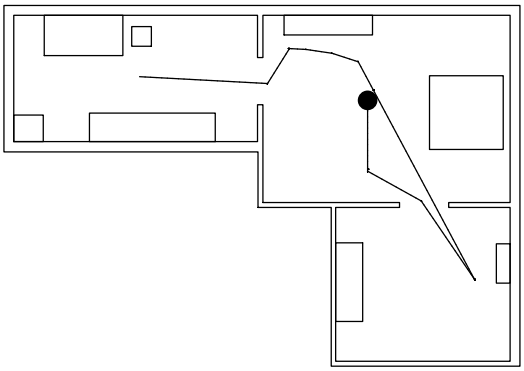


Fig. 3. Typical path taken by one of the best robots. The filled circle marks the initial position of the robot. In the particular case illustrated here, the behavior selection system was optimized using method M_1 and was obtained after the evaluation of 450 individuals.

As seen from the results in Table I, the value of $\bar{f}$ in methods M_1 and M_5 is similar to that of the SUF method whereas M_2, M_4, and M_6 perform significantly worse (on average). M_3 achieved intermediate values of $\bar{f}$.

In terms of the maximum fitness attained in any of the 10 runs ($f_{\max} = \max_i f_i^{\text{best}}$), all methods performed approximately equally well, i.e. all methods were capable of finding at least one good solution. If the EAs were allowed to run for longer periods of time (more than 5 000 evaluated individuals), all methods eventually managed to escape any local optima. The path travelled by one of the best robotic brains obtained using the M_1 method is shown in Fig. 3.

TABLE I

RESULTS FROM RUNS USING THE STANDARD UF (SUF) METHOD AND THE SIX METHODS DEFINED IN SECT. IV-B (M_1–M_6). A TOTAL OF 10 RUNS WERE MADE FOR EACH METHOD. IN EACH RUN, A TOTAL OF 5 000 INDIVIDUALS WERE EVALUATED. $f_{\max}$ IS THE MAXIMUM FITNESS ACHIEVED, $\bar{f}$ IS THE AVERAGE OF THE MAXIMUM FITNESS OVER THE 10 RUNS, AND $C_{95\%}$ IS THE CORRESPONDING 95% CONFIDENCE INTERVAL.

Method	$f_{\max}$	$\bar{f}$	$C_{95\%}$
SUF	59.07	55.75	1.92
M_1	58.52	54.61	2.00
M_2	54.20	40.87	8.53
M_3	55.92	49.63	5.01
M_4	58.36	40.96	10.00
M_5	58.15	55.17	1.36
M_6	54.87	40.86	6.87

VI. DISCUSSION

Looking at the averages $\bar{f}$ of the best results obtained in the 10 runs for each method, the most striking observation is the fact that the methods M_2, M_4, and M_6 perform significantly worse than the SUF method. By contrast, M_1, M_3, and M_5 which, like the SUF method, all start from complete polynomials, are comparable in performance to the SUF method (with the possible exception of M_3, which performs slightly worse).

This was not anticipated since M_2, M_4, and M_6 all start from utility functions with fewer polynomial terms than the SUF method and thus, effectively, are able to carry out their search in a lower-dimensional space, and with flexibility as to which terms should be included in the utility function polynomials. Having obtained these results, one might be tempted to conclude that all polynomial terms are needed in order to find a good solution or, in other words, that only complete polynomials, as defined in Sect. I, will lead to proper behavior selection. However, this is *not* the case since it is *possible* (albeit slower) to find good solutions even if the polynomial degree d is lowered to 2, say.

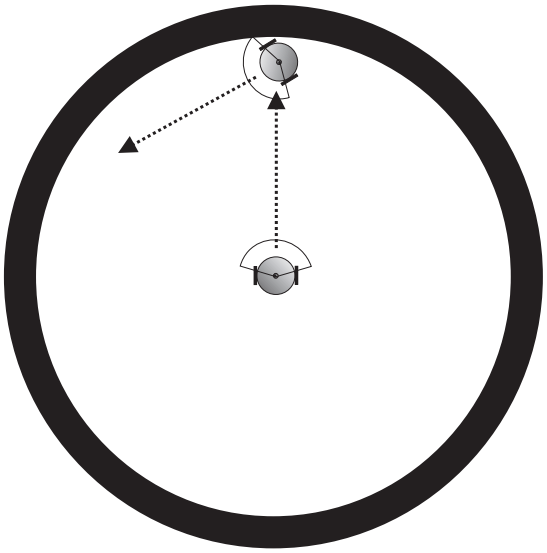


Fig. 4. A robot in an arena with a circularly symmetric obstacle. The robot is differentially steered and equipped with a single wide-range proximity sensor.

How, then, can the difference in performance be accounted for? Consider the simple toy problem in which a robot is placed in a circularly symmetric arena as shown in Fig. 4. The robot is assumed to be equipped with the two behaviors *straight-line navigation* (B_1) in which both motors receive equal signals, and *turning* (B_2), in which the motor signals are equal in magnitude but have opposite sign, making the robot turn without moving its center-of-mass. It is further assumed that the robot moves slowly, that it can stop almost instantaneously, and that it has an unlimited energy supply. The proximity sensor is assumed to give a signal $s = 1$ if an obstacle is detected and $s = 0$ otherwise. In order for the robot to achieve collision-free navigation, maximizing the distance travelled in a given time, it should, of course, keep B_1 active unless the sensor detects an obstacle, in which case the robot should turn until the obstacle is no longer detected, at which point the execution of B_1 should be resumed (see Fig. 4). Now, in order to solve this simple problem using the UF method, one can without restriction set one of the utility functions (U_1, say) to 0. For U_2, an ansatz of the form

$$U_2 = a_0 + a_1 s + a_2 s^2 + \ldots + a_d s^d, \tag{10}$$

can be used. For this simple problem, $d = 1$ would be sufficient. However, assuming that the level of complexity

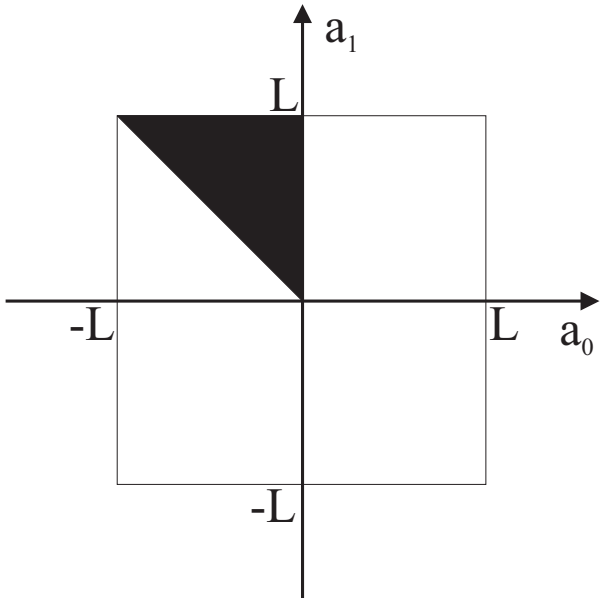


Fig. 5. The integral I_2. The region C corresponds to the shaded part of the space.

of the problem is unknown (as is normally the case, in more realistic applications), one must select a suitable value of d, and then search the space of possible solutions. The question is, then: What fraction of the search space will give correct solutions (as described above), as a function of the dimensionality ($D = d + 1$) of the search space? In this problem, correct solutions occur if $U_1 > U_2$ for $s = 0$ and $U_2 > U_1$ for $s = 1$. Hence, the conditions

$$a_0 < 0 \tag{11}$$

and

$$\sum_{i=0}^{d} a_i > 0 \tag{12}$$

must be satisfied in order for a given solution (i.e. a set of polynomial coefficients) to be correct. Assuming that all parameters a_i lie in the interval $[-L, L]$, the fraction of the search space meeting these criteria can be formulated as an integral, according to

$$I_D = (2L)^{-D} \int_{\{a_i\} \in C} \mathrm{d}A \ , \tag{13}$$

where $\mathrm{d}A$ denotes $\mathrm{d}a_0 \mathrm{d}a_1 \cdots \mathrm{d}a_d$, and the region C is defined by the criteria given in (11) and (12). As an example, the integral I_2 is illustrated in Fig. 5. As is evident from the figure, $I_2 = 1/8 = 0.125$. It is straightforward to compute the integral for any value of D, and the resulting values are shown in Table II. As is evident from the table, the fraction I_D of the search space giving correct solutions *increases* as the number of dimensions is raised. Put differently, the results in Table II exemplify the fact that an increase in the size of the search space does not always make it more difficult to find the correct solution. Instead it is the *fraction* of the search space containing correct solutions that matters.

TABLE II

THE INTEGRAL I_D SHOWN FOR VARIOUS VALUES OF THE DIMENSION (D) OF THE SEARCH SPACE. NUMERICAL VALUES WERE CALCULATED USING A MONTE CARLO METHOD WITH A TOTAL OF AROUND 10^9 SAMPLED POINTS FOR EACH VALUE OF I_D.

D	I_D	D	I_D	D	I_D
2	0.1250	6	0.1783	10	0.1948
3	0.1458	7	0.1837	11	0.1974
4	0.1615	8	0.1881	100	0.2327
5	0.1711	9	0.1918	214	0.2382

Finally, for the simulations, it should be noted that the maximum attained fitness values $f_{\max}$ shown in Table I, are quite similar. This is due to the fact that, during the 100 s simulations used here, there is a limit to the amount of fitness that can be attained. An estimate of the maximum attainable fitness (the details of which will not be given here) leads to a value of around 64, rather close to the maximum fitness values reached by all methods. Had the problem been more complex, involving, say, five or 10 behaviors, it is likely that the difference in performance between, on the one hand the SUF method, M_1, M_3, and M_5 and, on the other hand, M_2, M_4, and M_6 would be evident also in the values of $f_{\max}$.

VII. CONCLUSION

The main conclusion of this work is that, at least for certain problems with fairly low level of complexity, the performance of the standard UF method is equal to, or better than, the performance of the modified methods, despite the more rigid structure of the utility function polynomials in the SUF method. A possible explanation is the fact that, in some problems, an effective increase in the size of the search space does *not* necessarily make the problem of finding good solutions harder. This has been illustrated by means of an analytically tractable toy problem, for which it was shown that the probability of finding a correct solution *increases* with the size of the search space.

VIII. ACKNOWLEDGMENTS

The authors gratefully acknowledge financial support from the Carl Trygger foundation for this project.

REFERENCES

[1] R. Arkin, *Behavior-based robotics*. MIT Press, 1998.
[2] P. Pirjanian, "Behavior coordination mechanisms – state-of-the-art," Institute of Robotics and Intelligent Systems, USC, Los Angeles, Tech. Rep., 1999.
[3] M. Wahde, "A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions," *Journal of Systems and Control Engineering*, no. 217, pp. 249–258, 2003.
[4] http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary/Demo.html.
[5] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *Lecture notes in computer science*, vol. 929. Springer-Verlag GmbH, 1995, pp. 704–720.
[6] M. Wahde, J. Pettersson, H. Sandholt, and K. Wolff, "Behavioral selection using the utility function method: A case study involving a simple guard robot," in *Proc. of the* 3rd *Int. Symp. on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, 2005, pp. 261–266.

Paper VIII

Improving generalization in a behavioral selection problem using multiple simulations

in

Proceedings of the Joint 3rd International Conference on Soft Computing and Intelligent Systems and the 7th International Symposium on advanced Intelligent Systems (SCIS & ISIS 2006), Tokyo, Japan, September 2006, pp. 989–994.

Improving generalization in a behavior selection problem using multiple simulations

Jimmy Pettersson and Mattias Wahde

Department of Applied Mechanics
Chalmers University of Technology
412 96 Göteborg, Sweden
E-mail: {jimmy.pettersson, mattias.wahde}@chalmers.se

***Abstract*— Various methods of improving the validation performance of behavior selection systems for autonomous robots, based on the utility function (UF) method is investigated by means of simulations involving a robot carrying out a simple exploration task. In the UF method, behavior selection is based on utility functions which, in turn, are optimized using an evolutionary algorithm (EA). The computer simulations used for generating the utility functions can be set up in many different ways and, in this paper, several different setups are compared with respect to their ability of generating reliable behavior selection systems.**

The results show that better validation performance is obtained in cases where several simulations are carried out in the evaluation of an individual, provided that the simulations are sufficiently long so that the robot will have to make many behavior selection decisions. Furthermore, the results indicate that the choice of method for combining the results obtained in different simulations to form a scalar fitness value has only a rather limited impact on the performance.

I. Introduction and motivation

In order for autonomous robots to become widely applicable they must be able to operate in a great variety of different environments with varying degrees of unpredictability. In the behavior-based approach, robots are commonly equipped with a repertoire of (rather simple) behaviors and a method for selecting between those behaviors, also called a behavioral organization system. Due to its central importance, many methods have been developed for behavioral organization [1] or behavior selection, as it is also called. In this paper, the investigation will be centered around one such method, namely the utility function (UF) method [2], in which the selection of behaviors (for activation) is based on utility functions which, in turn, are optimized in simulations using an evolutionary algorithm (EA).

In the UF method, the evaluation of robotic brains (control systems) is based on the results obtained from one or several simulations, where simulations differ in their setup, i.e. the initial position and velocity of the robot, the initial battery energy, the number of moving obstacles (and their types of motion) etc. In cases where only a single training setup is used, the overall behavior of an evolved robot will invariably be tailored to that particular setup (i.e. overfitting will occur) limiting the ability of the robot to handle situations that were not encountered during training.

An additional problem is that detailed simulation of autonomous robots is a computationally very intensive undertaking: The simulator must, in each time step, take into account the sensors and actuators of the robot, as well as the interaction of the robot with its environment. In order to test the selection of behaviors in all relevant situations, each evaluation must usually cover a rather long time. Energy maintenance is a case in point: If a robot starts its operation with a near-full battery, it will clearly take a while before the battery charging behavior becomes relevant. Of course, this particular problem can be mitigated in several ways, e.g. by introducing an artificially high battery discharge rate, or by starting the robot with a near-empty battery. However, similar considerations occur in other behavior-switching situations as well.

In addition, other more subtle problems occur for the special case of behavior selection. For example, a robot may have a choice between say, a straight-line navigation behavior (B_1) and an obstacle avoidance behavior (B_2). As an illustration, consider the slightly simplified situation where the robot is started with a long, empty stretch in front of it, and assume that the robot is given fitness only for executing the navigation behavior, the obstacle avoidance behavior being an auxiliary behavior that does not contribute to the fitness of the robot [3]. The EA will then quickly find solutions in which the robot moves in a straight line and then hits a wall, causing the simulation to be terminated. After some time, robots will appear that are able to avoid the collision (for example by lowering the utility of B_1, or raising the utility of B_2 when a collision with a wall is imminent, as measured e.g. by IR sensors on the robot). However, the search problem is made more difficult by the absence of a proper gradient: Robots that follow a straight line into the wall will all receive the same fitness, regardless of how close they were to switching between the two behaviors. It is only when a switch actually occurs that the robot will be able to avoid the wall and then continue along some other direction, thus increasing its fitness. Of course, in this simple example, it would be possible to provide a gradient for the EA simply by letting the fitness measure also depend on how close the robot was to switching to the obstacle avoidance behavior. However, in most cases, it is undesirable to have to specify such a detailed fitness measure. In fact, one of the main advantages with the UF method is that the user should *not* have to provide a very detailed fitness measure involving all possible situations that may occur.

Clearly, this problem is not limited to the UF method - in any method where a reward (i.e. a fitness increase) is given only for the continuous execution of a single behavior,

the same problem will appear. Evidently, these problems, in connection with the long evaluation times, make the search for good behavior selection systems even more difficult (and time-consuming).

While the problem of overfitting (i.e. lack of generalization properties due to adaptation to special conditions) has been studied extensively for e.g. artificial neural networks, the problem has not been much studied in relation to behavior selection in autonomous robots. This paper aims to investigate how evaluations of robotic brains should be devised in order to evolve a behavior selection system capable of generalizing to previously unseen situations.

In Sect. II a brief introduction to the UF method is given. The simulations (setup and software) are described in Sect. III, and the results are presented in Sect. IV. In Sect. V, the results are discussed and some conclusions are drawn.

II. The utility function method

In the UF method, each behavior in the behavioral repertoire $\{B_1, B_2, \ldots, B_N\}$ is associated with a utility function that measures the relative merit of each behavior. By default, the UF method uses a polynomial ansatz for each utility function with the polynomial degree specified by the user. An example of a second degree polynomial ansatz is

$$\begin{aligned} U(s_1, p_1) &= a_{00} + a_{10}s_1 + a_{01}p_1 + a_{20}s_1^2 + \\ &+ a_{11}s_1p_1 + a_{02}p_1^2 \,, \end{aligned} \tag{1}$$

where s_1 and p_1 are state variables and the a_{ij} are constants to be determined by the EA. State variables can be of three kinds: (1) external variables, denoted s_i (e.g. the readings of IR sensors) (2) internal physical variables, denoted p_i (e.g. the battery level), and (3) internal abstract variables, denoted x_i (variables used as internal signals, roughly corresponding to hormones in biological organisms). As exemplified in Eq. (1), the UF method uses a complete polynomial, i.e. including all terms up to the specified degree, for each utility function.

The UF method is an arbitration method, i.e. a method in which only a single behavior is active at any time [1], and it uses a rather straightforward behavior selection mechanism: At all times, the behavior currently associated with the highest utility value (as generated by the corresponding utility function) is selected for activation. For a more detailed description of the UF method, see [2].

As in any application involving an EA, a proper assignment of fitness is crucial in order for the robot to carry out its intended task. In the UF method, the assignment of fitness is often associated with the execution of a given *task* behavior, during which the fitness is changed (increased). Other behaviors, not directly related to the designated task, are referred to as *auxiliary* behaviors. For instance, if a robot is supposed to follow walls, the behavior *follow wall* is the designated task behavior and the fitness measure could, for instance, be proportional to the distance traveled during the active period of that behavior. Continuing the same example, in order to avoid battery depletion, the robot's behavior selection system must activate the behavior *charge battery* every now and then in order to secure future fitness increases (by making it possible for the robot to travel further). Since the behavior *charge battery* is not directly linked to the wall following task, it is considered to be an auxiliary behavior.

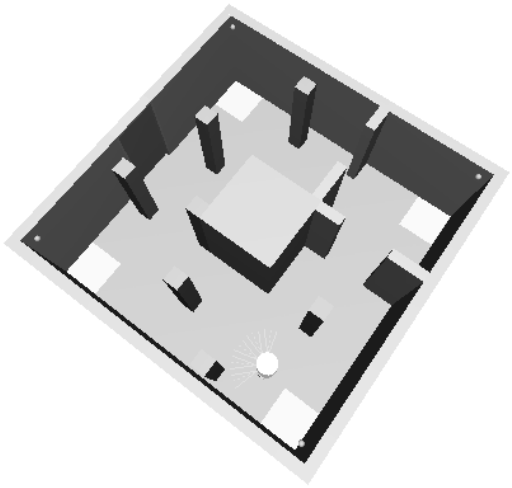


Fig. 1. Arena used in the simulations (seen from above).

III. Simulations

Simulations were made in software built using the UFLibrary software package [4]. The UFLibrary is intended to provide a quick and reliable way of generating behavior selection systems for any type of agent (e.g. an autonomous robot). Implemented in object-oriented Pascal, it provides, in addition to the basic framework of the UF method, the equations of motion for differentially steered robots, a numerical solver, collision managers, OpenGL visualization, and various sensor models.

A. Simulation setup

The task of the robot was to explore the arena shown in Fig. 1, corresponding to e.g. a room in a gallery, containing a number of obstacles and four charging stations placed in the corners of the arena (indicated in Fig. 3). In order to make it easier for the robot to locate the charging stations, each station was accompanied by an IR beacon, transmitting a signal that the robot could identify.

The robot model was chosen to be of a differentially steered type, with a cylindrical shape. The robot was equipped with five IR sensors positioned symmetrically on the front of the robot's body, each having a range of 0.5 m and an opening angle of 0.5 radians. The robot was also able to determine the amount of energy stored in its onboard battery, which discharges as

$$\frac{\mathrm{d}E}{\mathrm{d}t} = -k_r - k_m|v| \,, \tag{2}$$

where k_r and k_m are positive constants and v is the velocity of the robot's center of mass. The constants were set so that a full battery would last approximately 50 s. Assuming that the robot is positioned at a charging station, battery charging occurs at a constant rate, allowing a fully depleted battery to recharge in 10 s. The unrealistically short intervals for discharging and charging were, of course, selected so that the robot would

have to activate its battery charging behavior from time to time, even in the rather short simulations used in this paper.

In order for the robot to be able to perform the task of exploring the arena, it was equipped with five behaviors, arranged in a hierarchical structure shown in Fig. 2. At the top level there are three behaviors: (1) *straight-line navigation* (B_1), (2) *obstacle avoidance* (B_2), and (3) *energy maintenance* (B_3). Behavior B_3 is divided into the two behaviors *charging station localization* ($B_{3.1}$) and *battery charging* ($B_{3.2}$). In UFLibrary [4], behaviors are compared on a level-by-level basis and the hierarchy is traversed until a single behavior is encountered. Thus, for each time step in the simulations, the utility values associated with the behaviors B_1, B_2, and B_3 are compared first. In case B_3 has the highest utility, another comparison is made between $B_{3.1}$ and $B_{3.2}$, resulting in the selection of one of them. If the utility values associated with $B_{3.1}$ and $B_{3.2}$ happen to be equal in magnitude, the first behavior in the hierarchy (in this case $B_{3.1}$) is selected for activation.

Based on earlier work [5], a polynomial degree of three was chosen for the five utility functions, which were given the following functional form

$$\left.\begin{array}{rcl} U_1 & = & U_1(s_1, s_2, s_3, s_4, s_5, p_1) \\ U_2 & = & U_2(s_1, s_2, s_3, s_4, s_5, p_1, x_1) \\ U_3 & = & U_3(x_2, p_1) \\ U_{3.1} & = & U_{3.1}(s_6) \\ U_{3.2} & = & U_{3.2}(x_3, p_1) \end{array}\right\} \quad (3)$$

where s_1–s_5 are the readings from the five IR sensors, p_1 is the amount of energy in the robot's battery, x_1 and x_2 are internal abstract variables corresponding to *fear*, and *hunger*, respectively. s_6 is the beacon detector signal, and x_3 is an internal abstract variable corresponding to *inverse satiation*. Internal abstract variables (x_i) were used in a simplified, binary manner where the value of each x_i was set to 1 during the active period of the associated behavior and 0 otherwise.

The beacon detector (on the robot) was given an opening angle of 0.08 radians and the magnitude of the generated signal ($s_6 \in [0, 1]$) was calculated as $\min(d^{-2} \cos\alpha, 1)$, where d is the distance from the robot to the beacon, and α is the relative angle (a detection occurred only if $|\alpha| < 0.04$).

A total of four beacons were present in the arena, each positioned at a charging station (see Fig. 3). Also, a beacon could only be detected if there was an unobstructed line of sight between the beacon and the detector.

Since the task of the robot was to cover as much ground as possible, the designated *task* behavior (see Sect. II) was B_1, during which the motor signals were set to equal values, causing the robot to move forward in an asymptotically straight line. The fitness for a given simulation was taken proportional to the time spent in B_1 and upon exit from that behavior, a fitness increase of $\max(0, t_1 - 1)$, where t_1 is the time spent in B_1, was given. By only rewarding active periods longer than one second, behavior dithering [6], was effectively removed.

In B_2, the robot's heading is changed until the IR sensors indicate that there are no obstacles in front of the robot and the robot then stops. B_3 simply delegates control over the robot either to $B_{3.1}$, in which the robot tries to locate a battery charging station by detecting a beacon signal and then approaching it, or to $B_{3.2}$, in which the robot stops in order to charge its battery. Although $B_{3.2}$ may be selected for activation by the behavior selection system at any time, actual charging of the battery only occurs if the robot is positioned at one of the charging stations.

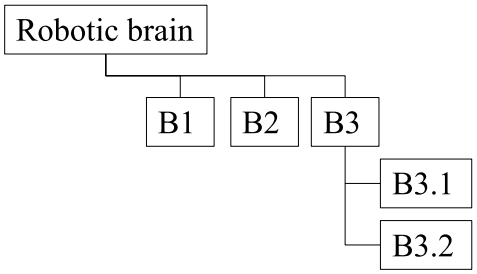


Fig. 2. Behavioral hierarchy used in the behavior selection system.

B. Evaluation setup

In order to investigate how different fitness measures, simulation times, and size of the training set affected the robot's ability to generalize, a number of evaluation setups were defined (see Table II). Each evaluation setup consists of several simulations (see Sect. III-A), which, in turn, belong either to a training set or to a validation set. For the training set, combined fitness values were calculated in one of three ways as

$$F_a = \frac{1}{N_T} \sum_{i=1}^{N_T} f_i, \quad (4)$$

$$F_b = \min(f_1, f_2, \ldots, f_{N_T}), \quad (5)$$

$$F_c = F_b + \varepsilon F_a \quad (6)$$

where N_T is the size of the training set (the number of simulations), f_i is the fitness achieved in the i^{th} training simulation, and ε is a small positive constant. The first fitness measure, F_a, considers the average performance of the robot during its training simulations, whereas the second measure F_b only counts the *worst* performance. The use of F_b is intended to lead the EA away from solutions that only work on average. However, when F_b is used, it may be difficult for the EA to find a way forward, since a great deal of information concerning the performance of the robot is removed in the forming of the fitness value. The third fitness measure, F_c, aims to combine F_a and F_b, by focusing on the worst result while also considering the average: The second term in the equation for F_c makes it possible for the EA to distinguish between evaluations with identical worst simulations but different averages.

In each of the N_T training simulations, the robot was placed at a given, fixed position, and the initial direction of the robot was also specified deterministically, see Sect. IV below. The initial energy was set to 1.0 in all training simulations[1].

In order to test the ability of the behavior selection system to generalize and to make comparisons possible, a validation set consisting of 10 different simulations (each with a maximum

[1]The battery level is given as a dimensionless value with 1.0 meaning a fully charged battery.

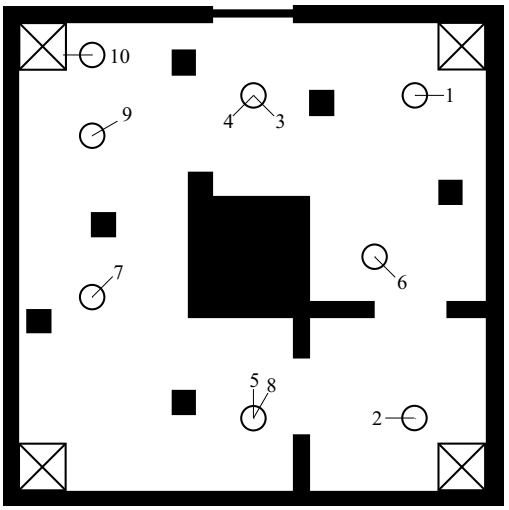


Fig. 3. Circles mark the initial position of the robot in each of the 10 validation setups. Lines emanating from the circles indicate the initial heading of the robot. Note that some validation simulations (e.g. 5 and 8) use the same initial position, but different headings. Crosshatched areas mark the locations of the four charging stations.

length of 100 s) was also defined. None of the validation simulations were used in the computation of the fitness value. Instead, the 10 validation simulations were executed every time a new best individual was found, i.e. one with a higher fitness value (over the training simulations), as defined by the combined fitness measure (F_a, F_b, or F_c), than the previous best individual. The results $f_i^{\text{val}},\ i = 1, \ldots, 10$ over the validation simulations were recorded but, as mentioned above, were *not* used to guide the EA in its search. The initial positions and headings of the robot in the validation simulations are shown in Fig. 3. The initial battery level was set to 1.0, 0.2, 0.5, 0.3, 1.0, 0.6, 0.8, 0.8, and 0.1, in validation simulations one to 10, respectively.

C. Evolutionary algorithm

The EA used for the optimization of the behavior selection systems was a genetic algorithm (GA), with a single, fixed-size population and employing generational replacement, i.e. the formation of new individuals was performed in such a way that all evaluated individuals were replaced by new individuals in a single step. New individuals were formed through a sequence of tournament selection, single-point crossover, and parameter mutation. Elitism was used, i.e. an exact copy of the current best individual was transferred, unchanged, to the next generation.

The parameter values (common to all runs) for the utility functions and the EA are shown in Table I. n_c refers to the number of coefficients in the utility function polynomials which, using the functional form defined in Eq. (3), was equal to 228 [5].

IV. Results

In each run of the EA, a total of $10,000$ individuals were evaluated. The population size was set to 100, and the number of generations was therefore also equal to 100. Due to the stochastic nature of EAs, i.e. the fact that the initial population is generated randomly, and the fact that the formation of new individuals involves stochastic computations as well, a total of 5 runs were made for each evaluation setup.

TABLE I

Parameter settings common to all runs (B and E_1-E_9). Note that the total simulation time refers to the training simulations. In the 10 validation simulations, each simulation lasted a maximum of 100 s, see Subsect. III-B.

Time step length	0.01 s
Maximum *total* simulation time	100 s
Polynomial degree	3
Number of validation simulations	10
Tournament selection probability	0.75
Crossover probability	0.5
Mutation probability	$4/n_c$

TABLE II

Specification of the benchmark setup (B) and the additional setups E_1 - E_9. N_T denotes the number of simulations used per evaluation, and $T_{\max}$ is the maximum time for each simulation. The fourth column specifies the fitness measure used. In cases where F_c was used, the parameter ε was set to 0.001.

Setup	$T_{\max}(s)$	N_T	Fitness measure
B	100.00	1	F_a
E_1	50.00	2	F_a
E_2	50.00	2	F_b
E_3	50.00	2	F_c
E_4	33.33	3	F_a
E_5	33.33	3	F_b
E_6	33.33	3	F_c
E_7	10.00	10	F_a
E_8	10.00	10	F_b
E_9	10.00	10	F_c

First, a set of benchmark runs were carried out, using a single training simulation of maximum length $T = 100$ s, and an initial position (of the robot) close to the lower left corner in Fig. 3.

The specification of the benchmark runs (B) and of the other runs (E_1 - E_9), which used different evaluation setups, are shown in Table II. In runs E_1 - E_9, several training simulations were used in the evaluation of individuals, and one of the combined fitness measures F_a, F_b, or F_c was applied. In order to make a fair comparison between runs, the total maximum evaluation time (for training) was kept fixed at 100 s. Thus, in runs with, say, three training simulations per evaluation, each simulation lasted a maximum of 33.33 s. However, note that, in case of collisions between the robot and an obstacle, the simulation was terminated immediately. Thus, the actual simulation time was often shorter than the stipulated upper limit.

In all runs, the starting positions of the robot in the *training* simulations were the same as in the benchmark run, but different directions were chosen for the N_T simulations. Thus, for example, two initial directions were selected for E_1, and were then used for *all* individuals in the E_1 runs (and the same two directions were also used in runs E_2 and E_3). For E_4 - E_6, three different directions were chosen etc. The results are presented in Tables III and IV. In Table III, the detailed results from each of the $10 \times 5 = 50$ runs are shown. Whereas the fitness measures differed between runs, the validation simulations were, as indicated above, identical in all runs, and

TABLE III

DETAILED RESULTS OBTAINED FOR THE BENCHMARK RUNS (B) AND FOR THE RUNS USING EVALUATION SETUPS E_1 - E_9. THE SYMBOLS USED IN THE TABLE ARE DESCRIBED IN DETAIL IN CONNECTION WITH EQ. (7) IN THE MAIN TEXT.

Run	$\overline{f}_{\rm val}(10)$	$\overline{f}_{\rm val}(30)$	$\overline{f}_{\rm val}(100)$	$\overline{f}_{\rm val}^{\rm max}$	$j_{\rm val}^{\rm max}$	Run	$\overline{f}_{\rm val}(10)$	$\overline{f}_{\rm val}(30)$	$\overline{f}_{\rm val}(100)$	$\overline{f}_{\rm val}^{\rm max}$	$j_{\rm val}^{\rm max}$
B_1	7.826	6.567	8.310	11.973	2385	$E_{5,1}$	8.608	10.792	12.148	14.797	738
B_2	5.459	8.221	6.589	13.200	5490	$E_{5,2}$	8.452	6.516	7.342	13.654	2048
B_3	8.679	6.674	8.975	12.674	4556	$E_{5,3}$	17.335	12.273	18.992	19.008	4579
B_4	8.394	8.967	7.516	10.005	1533	$E_{5,4}$	10.373	10.847	15.860	15.910	8973
B_5	7.844	14.466	10.664	16.602	3578	$E_{5,5}$	4.913	14.011	12.464	21.164	4333
$E_{1,1}$	6.180	14.059	14.722	19.149	1924	$E_{6,1}$	9.687	12.236	15.727	15.727	9500
$E_{1,2}$	5.609	6.579	6.256	12.179	7378	$E_{6,2}$	7.064	11.275	11.531	11.531	7308
$E_{1,3}$	6.364	19.301	15.548	20.974	3115	$E_{6,3}$	14.568	10.187	7.340	17.224	740
$E_{1,4}$	4.747	15.684	17.007	18.739	3212	$E_{6,4}$	10.652	8.352	10.980	13.212	5427
$E_{1,5}$	17.005	17.067	15.825	20.375	1765	$E_{6,5}$	6.304	5.668	8.821	15.267	3543
$E_{2,1}$	7.754	10.721	7.628	13.106	5320	$E_{7,1}$	13.667	9.633	9.897	19.436	4911
$E_{2,2}$	7.679	7.173	6.779	8.649	1486	$E_{7,2}$	2.878	2.476	3.184	10.247	240
$E_{2,3}$	8.219	7.075	7.206	13.447	202	$E_{7,3}$	8.983	16.760	12.091	22.700	5829
$E_{2,4}$	4.707	18.567	18.619	19.799	5607	$E_{7,4}$	3.470	4.966	6.947	7.048	4941
$E_{2,5}$	5.978	14.240	16.409	19.385	8551	$E_{7,5}$	7.704	9.828	6.793	11.656	1430
$E_{3,1}$	11.761	13.556	11.365	20.725	7210	$E_{8,1}$	2.495	3.687	2.737	5.238	1948
$E_{3,2}$	12.446	5.911	9.239	14.106	9107	$E_{8,2}$	3.660	4.943	3.997	5.962	1305
$E_{3,3}$	12.991	13.111	17.889	20.705	7460	$E_{8,3}$	7.716	6.748	10.422	15.948	3602
$E_{3,4}$	7.854	16.011	16.006	22.729	7150	$E_{8,4}$	8.480	7.571	8.569	11.384	7236
$E_{3,5}$	9.503	13.997	9.449	16.663	3225	$E_{8,5}$	9.412	13.761	9.504	15.405	2254
$E_{4,1}$	18.257	15.145	12.664	18.257	816	$E_{9,1}$	7.704	9.828	6.793	11.656	1430
$E_{4,2}$	13.511	10.947	17.025	17.268	5603	$E_{9,2}$	9.001	12.178	8.386	14.792	1994
$E_{4,3}$	5.582	10.607	11.705	20.578	5772	$E_{9,3}$	10.363	8.510	7.330	11.460	1163
$E_{4,4}$	10.478	10.045	8.440	12.377	5472	$E_{9,4}$	8.826	10.191	5.458	13.934	1757
$E_{4,5}$	11.000	9.612	14.524	18.435	8226	$E_{9,5}$	13.935	6.228	7.998	17.956	1293

the results of those simulations can therefore be compared directly. The average validation performance of an individual was defined as

$$\overline{f}_{\rm val} = \frac{1}{10}\sum_{i=1}^{10} f_i^{\rm val}, \tag{7}$$

with $f_i^{\rm val}$ defined as discussed in Subsect. III-A. In Table III, the symbol $\overline{f}_{\rm val}(k)$ indicates the average validation performance of the best individual in generation k, i.e. the individual with the highest fitness over the *training simulations*, as defined by the combined fitness measure, see Table II. $\overline{f}_{\rm val}^{\rm max}$ signifies the best average validation performance found in the run, and $j_{\rm val}^{\rm max}$ indicates which individual (among the $10,000$ evaluated individuals) achieved this result.

In Table IV, the results are summarized in abbreviated form. Here, averages have been taken also over the five runs carried out for each evaluation setup. $\overline{\overline{f}}_{\rm val}(k)$ thus denotes the arithmetic average, over five runs, of $\overline{f}_{\rm val}(k)$, and $\overline{\overline{f}}_{\rm val}^{\rm max}$ is defined analogously.

TABLE IV

SUMMARY OF THE RESULTS OBTAINED FOR THE BENCHMARK RUNS (B) AND FOR EVALUATION SETUPS E_1 - E_9. FOR EACH EVALUATION SETUP, THE SECOND, THIRD, AND FOURTH COLUMNS SHOW THE AVERAGE (OVER THE FIVE RUNS) OF THE AVERAGE (OVER THE 10 VALIDATION SIMULATIONS) VALIDATION PERFORMANCE FOR THE BEST INDIVIDUAL IN GENERATIONS 10, 30, AND 100, RESPECTIVELY. THE FIFTH COLUMN SHOWS THE AVERAGE (OVER THE FIVE RUNS) OF THE BEST VALIDATION PERFORMANCE OBTAINED.

Setup	$\overline{\overline{f}}_{\rm val}(10)$	$\overline{\overline{f}}_{\rm val}(30)$	$\overline{\overline{f}}_{\rm val}(100)$	$\overline{\overline{f}}_{\rm val}^{\rm max}$
B	7.640	8.979	8.411	12.891
E_1	7.981	14.538	13.872	18.283
E_2	6.867	11.555	11.328	14.877
E_3	10.911	12.517	12.790	18.980
E_4	11.766	11.271	12.872	17.383
E_5	9.936	10.888	13.361	16.907
E_6	9.655	9.544	10.880	14.592
E_7	7.340	8.733	7.782	14.217
E_8	6.353	7.342	7.046	10.787
E_9	9.966	9.387	7.193	13.960

V. DISCUSSION AND CONCLUSION

The results (e.g. $\overline{\overline{f}}_{\rm val}^{\rm max}$) presented in the previous section, and particularly in Table IV, indicate that the naive way of carrying out the simulations (i.e. using the benchmark setup, with a single training simulation) gives worse results than runs employing several, shorter training simulations, regardless of the type of combined fitness measure (F_a, F_b, or F_c) used. This conclusion applies to evaluation setups E_1 - E_6, in which either two simulations of length 50.00 s or three simulations of length 33.33 s were used in the evaluation of individuals. Thus, it appears that subjecting the robot to multiple fitness calculations indeed forces the EA to find behavior selection systems that operate in a more general way than those obtained in the benchmark runs, where the robot can do well by finding a single path, from one initial condition.

As is also evident from Table IV, however, the opposite occurs in cases where the number of training simulations is increased to 10 (evaluation setups E_7 - E_9). Here, the results are similar to those from the benchmark runs. Presumably, the reason for this is that the maximum simulation time (10.00 s) used in these runs is simply *too* short in order for the robot to encounter all relevant situations. For example, in such short simulations, there is little need for the robot to discover how to activate the energy maintenance behavior since, in most cases, its battery energy will be sufficient to survive for the duration of the simulation, without re-charging.

Thus, the main conclusion from the investigation is that results do improve if multiple training simulations are used, but also that improvements are only seen if the simulations are kept sufficiently long so that all relevant switches between behaviors can be tested. This conclusion can be strengthened by considering the performance of the robot on one of the most difficult validation simulations, namely the 10^{th} one, in which the robot starts with very low energy (0.10). For this simulation, rather mediocre results were found in all benchmark runs, where the robot would simply avoid activating B_3 and thus run out of energy quite fast, reaching fitness values of around 2 (with one exception, in which a fitness value of 8 was reached). By contrast, in evaluation setups E_1 - E_6 several cases were found in which the robot obtained fitness values of up to 19.8 for the 10^{th} validation simulation, even though, in fairness, such good results were quite rare even for these evaluation setups.

As mentioned in Subsect. III-B, in all training runs in setups B and E_1 - E_9 the initial energy of the robot was set to 1.0, i.e. to the largest possible value. This, in turn, may have further reduced the incentive for the robot to charge its battery. However, the validation results obtained in a few additional runs (e.g. $\tilde{\text{E}}_3$ in which the initial energy was set to 0.30) were, in fact, worse than those of the original runs. This also indicates, perhaps, that even though there is a greater incentive for the robot to activate energy maintenance in, say, $\tilde{\text{E}}_3$ than in E_3, it is quite difficult for the EA to find such solutions capable of doing so, possibly due to the choice of variables for the utility functions (see below).

Somewhat surprisingly, it can also be concluded from Table IV that the choice of combined fitness measure (F_a, F_b, or F_c) had very little effect on the results. In particular, the expected improvement for F_c (over F_a and F_b) did not materialize.

It should be noted that the number of runs (i.e. five) carried out for each evaluation setup is a little too small to allow a reliable, formal statistical analysis; since each run took around $100,000$ s to complete (on a 3 GHz computer), it was necessary to limit the number of runs to five for each setup. This is also the main reason why the runs were kept deterministic[2], in the sense that (1) the initial position and direction of the robot were *not* randomized, i.e. for a given evaluation setup, the starting position of the robot was fixed, as were the starting direction(s) used in the N_T simulations, and (2) no noise was applied, neither to sensor signals nor to motor signals.

While the absence of noise represents a considerable deviation from the case of real robots in which, for example, sensors are invariably noisy and the actual kinematics and dynamics do not correspond perfectly to the idealized models used here, the simplification is motivated by the fact that the aim of the paper has been to *compare* different ways of setting up evolutionary simulations rather than attempting to make as exact a representation of reality as possible. Assuming that the simulated sensors and the simulated kinematics and dynamics do not contain *systematic* deviations from what would be obtained with a real robot, one may consider the results of each simulation as representing an average of the results that would be found over many runs using a real robot. However, even if systematic deviations were present, the comparative analysis (i.e. the comparison of validation performance between different evaluation setups) would still be valid, even though the absolute fitness values obtained from any given simulated robot would differ, perhaps significantly, from those obtained with a real robot.

It is possible that the conclusions may have been slightly different had the initial position and heading of the robot been randomized. However, if, in addition to the stochasticity introduced by the EA, the initial conditions had been randomized, and noise had been introduced, an even larger number of runs would have been needed to draw any conclusions. Such an analysis is the topic of an investigation currently underway.

An additional difficulty concerns the specification of the structure of the robot brain, see Fig. 2, and the definition of the utility functions, both of which can, of course, be done in different ways; this investigation was limited to one configuration. It is possible, however, that the commonly seen failure of the robot to activate the energy maintenance behavior in e.g. the 10^{th} validation simulation was, in part, a result of the choice of state variables in the utility functions. More specifically, the utility function U_3 (which is directly compared to U_1 and U_2 in the selection of behaviors on the highest level in the hierarchy) depended only on two variables, rather than six or seven as for U_1 and U_2, respectively. This state of affairs *may* have made it more difficult for the robot to activate B_3 and the failure to do so would then not only be a result of the choice of evaluation setup. An investigation of this issue is also underway and, to conclude this paper, it should thus be noted that the form of comparison (i.e. of evaluation setups) attempted here is, in fact, more difficult to carry out than one would suppose at a first glance, due to the complexity of the problem.

[2]However, the EA did, of course, employ stochastic operators, as mentioned above.

VI. Acknowledgments

The authors would like to acknowledge financial support for this project from the Carl Trygger foundation.

References

[1] P. Pirjanian, "Behavior-coordination mechanisms – state-of-the-art," Institute for Robotics and Intelligent Systems, University of Southern California, Technical report IRIS-99-375, October 1999.

[2] M. Wahde, "A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions," *Journal of Systems and Control Engineering*, vol. 217, no. 4, pp. 249–258, September 2003.

[3] ——, "Evolutionary robotics," Tutorial, 2005, http://www.me.chalmers.se/~mwahde/AdaptiveSystems/Tutorials.html.

[4] http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary.

[5] M. Wahde, J. Pettersson, H. Sandholt, and K. Wolff, "Behavioral selection using the utility function method: A case study involving a simple guard robot," in *Proc. of the* 3^{rd} *Int. Symp. on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, 2005, pp. 261–266.

[6] B. M. Blumberg, "Action-selection in Hamsterdam: Lessons from ethology," in *From Animals to Animats 3: Proceedings of the* 3^{rd} *International Conference on Simulation of Adaptive Behaviour (SAB94)*. MIT Press, 1994.

Paper IX

A General-purpose Transportation Robot: An Outline of Work in Progress

in

The 15^{th} IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 06), Hatfield, United Kingdom, 2006, pp. 722–726

A General-purpose Transportation Robot
An Outline of Work in Progress

Mattias Wahde and Jimmy Pettersson

***Abstract*— An outline of a current joint project between Chalmers University of Technology (in Sweden) and several Japanese universities (Waseda University, Future University, and the University of Tsukuba) is presented. The aim of the project is to build a general-purpose transportation robot for use in hospitals, industries, and similar facilities. The project will provide a thorough test of the recently developed utility function method for behavior selection, which will be used for generating the decision-making system in the transportation robot.**

In this paper, an outline of the proposed transportation robot is given, along with a brief description of some of the challenges arising from this project. Furthermore, the utility function method is presented. Finally, the results obtained thus far are briefly discussed, and some directions for further work are provided.

I. Introduction

The combination of reduced hardware prices and the development of behavior-based (and related) techniques [2] has led to a rapid development of autonomous robots during the last two decades. Some of the tasks carried out by such robots include vacuum cleaning [16], entertainment [1] or general assistance to people, either at their place of work [7] or in their home [13], [15].

Another task that could potentially be carried out by robots is internal transportation (or delivery), i.e. the task of reliably moving objects from an arbitrary point A to another arbitrary point B in some (indoor) environment, without human supervision. Robots equipped with the means of carrying out such a task would be useful for internal transportation of various objects in hospitals, offices, or factories.

The development of a transportation robot is the main goal of a current joint project involving researchers at Chalmers University of Technology, in Göteborg, Sweden, Waseda University in Tokyo, University of Tsukuba, and Future University in Hakodate. Similar robotic platforms are being developed within the framework of other projects as well, e.g. the TUG robot [19], the Xavier robot [17], and the MB385 mobile transportation system [10].

While the definition of the problem may appear to be quite simple, the problem poses several difficult challenges that will be described in Sect. II below. The challenges pertain to hardware and software alike. On the hardware side, the construction of the robot and, in particular, the choice of an adequate set of sensory modalities must be considered carefully. On the software side, the problem of behavior selection is the main challenge. In Sect. III, a brief description of the utility function method for behavior selection will be given, and in Sect. IV, a brief discussion of the results obtained so far will be presented, along with a brief outline of future work.

Both authors are with the Dept. of Applied Mechanics, Chalmers Univ. of Technology, Goteborg, Sweden. Corresponding author: Mattias Wahde, mattias.wahde@chalmers.se

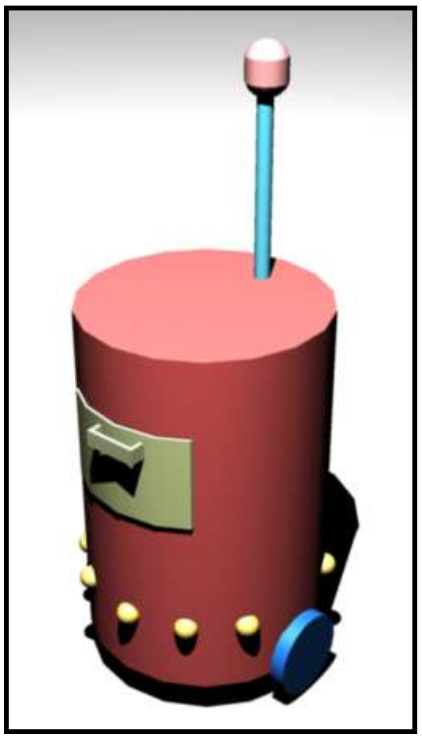


Fig. 1. A schematic illustration of the transportation robot. The laser range finder is located at the top of the pole.

II. Project outline

As mentioned above, the main goal of this project is to generate an autonomous robot capable of reliable internal transportation in indoor settings. An additional goal, however, is to test (and further develop) the utility function method for behavior selection. The transportation robot will constitute one of the first stringent tests of this method outside a laboratory setting.

A schematic drawing of the (differentially steered) robot is shown in Fig. 1. It is assumed that the brain of the robot[1] has been equipped with a map of the stationary parts

[1]The first prototype of the transportation robot will use a laptop computer. In later versions, a set of microcontrollers might be used.

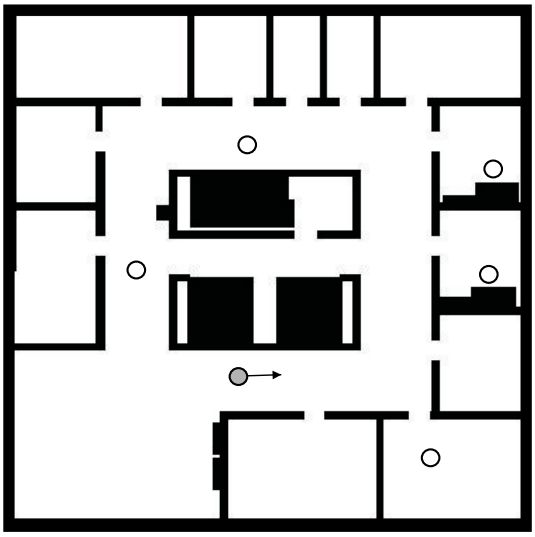


Fig. 2. An example of a typical environment for the transportation robot. The robot is shown as a filled circle, and the open circles indicate moving obstacles (e.g. people).

(e.g. the walls and doorways) of the arena. The sensory modalities involve (1) an array of IR sensors (or, possibly, a sonar array) for proximity detection, (2) a (2D) laser range finder, to be used for localization, in conjunction with digital optical encoders (one for each wheel), (3) a battery sensor, measuring the amount of energy available in the onboard battery, and (4) bumper sensors for detecting collisions. However, the robot will *not* be equipped with a GPS localization system[2]. Furthermore, it is assumed that the compartment of the robot used for transporting objects (hereafter: the transportation compartment) is equipped with scales, so that it can determine whether or not it is carrying an object[3].

An example of a typical arena for the transportation robot is shown in Fig. 2. The arena can represent, for example, a hospital ward, an office floor, or a factory. In this (schematic) figure, the robot is represented as a filled circle, and moving obstacles (e.g. people) as open circles. A brief description of a typical task for this robot will now be given.

A. Basic functionality

In a typical situation, the robot will start at (an arbitrary) point A, as indicated in the upper left panel of Fig. 3. A user will open the door to the transportation compartment, and place an object there. The robot will measure the weight of the object, giving a warning should the object be too massive. Next, the user will enter (via an, as yet unspecified, user interface) the intended navigation goal (point B) of the robot. The position of the navigation goal can possibly be given in the form of coordinates (x, y) or, more simply, chosen from a list of allowed positions, supplied to the robot in connection with the map. Possibly, for calibration, the robot may request information concerning its current position. Next, the robot will activate its *navigation* behavior (B1), generating a path towards its target location (point B), and begin moving. The path will be generated using the A* algorithm [6], which has been integrated with the UFLibrary software package, see Sect. III below. During the motion, the robot will constantly update its measured position through integration of the kinematic equations using the information supplied by the digital optical encoders. In addition, the robot will check continuously its immediate surroundings for obstacles. Should such an obstacle be detected in the direction of motion, the robot will suspend B1 and instead activate an *obstacle avoidance* behavior (B2). In B2, the robot will first stop moving in order to make sure that it does not collide e.g. with a person. Next, the robot will wait for a moment to see if the obstacle disappears. If it does not, the robot will then attempt to circumnavigate the obstacle, as indicated in the upper right panel of Fig. 3, again keeping track of its position, using the odometric readings. Once free of the (stationary or moving) obstacle, the robot will again activate B1, generating a new path towards point B, and resume its navigation.

Clearly, at some stage, the drift in the odometry will begin to pose problems. This is indicated in the lower left panel of Fig. 3, where the dashed circle indicates the position as perceived by the robot which, at this stage, differs from the actual position (indicated by the filled circle). Now, the robot should re-calibrate its odometric readings, and will thus activate an *odometry calibration* behavior (B3). The re-calibration will be carried out by matching the current readings of its laser range finder to the readings obtained at a given snapshot. Thus, a further assumption will be that a number of such laser range finder snapshots have been stored in advance, for example in connection with the storage of the map. The snapshots can either be in the form of a finite number of actual laser range finder readings, or in the form of estimates, for any point in the arena, based on the map. The former case is illustrated in the lower right panel of Fig. 3, where the snapshot points are indicated as small filled squares. Some of the rays of the laser finder are shown as well, as the robot attempts to match its current readings to those obtained at a nearby snapshot p.

Provided that the robot carries out the calibration with sufficient frequency (see Subsect. II-B below), it will only need to try to match its current location to the nearest snapshot. Once the (far from trivial) matching has been completed, the robot can again resume operation of its *navigation* behavior (B1).

Upon reaching the target location (point B), the robot will activate a *waiting* behavior (B4), in which it simply remains at standstill until a user removes the object it is carrying, and possibly gives the robot a new task.

Additionally, the robot will be equipped with an *emergency* behavior (B5) which can be activated if, despite its efforts to find point B, it finds itself stuck or lost.

[2]In general, the GPS signal is too weak to penetrate the walls of buildings. This problem can be solved, see e.g. [9], but here it will, nevertheless, be assumed that neither GPS nor any similar system for indoor applications is used.

[3]The maximum weight for objects transported by the robot will be around 20 kg.

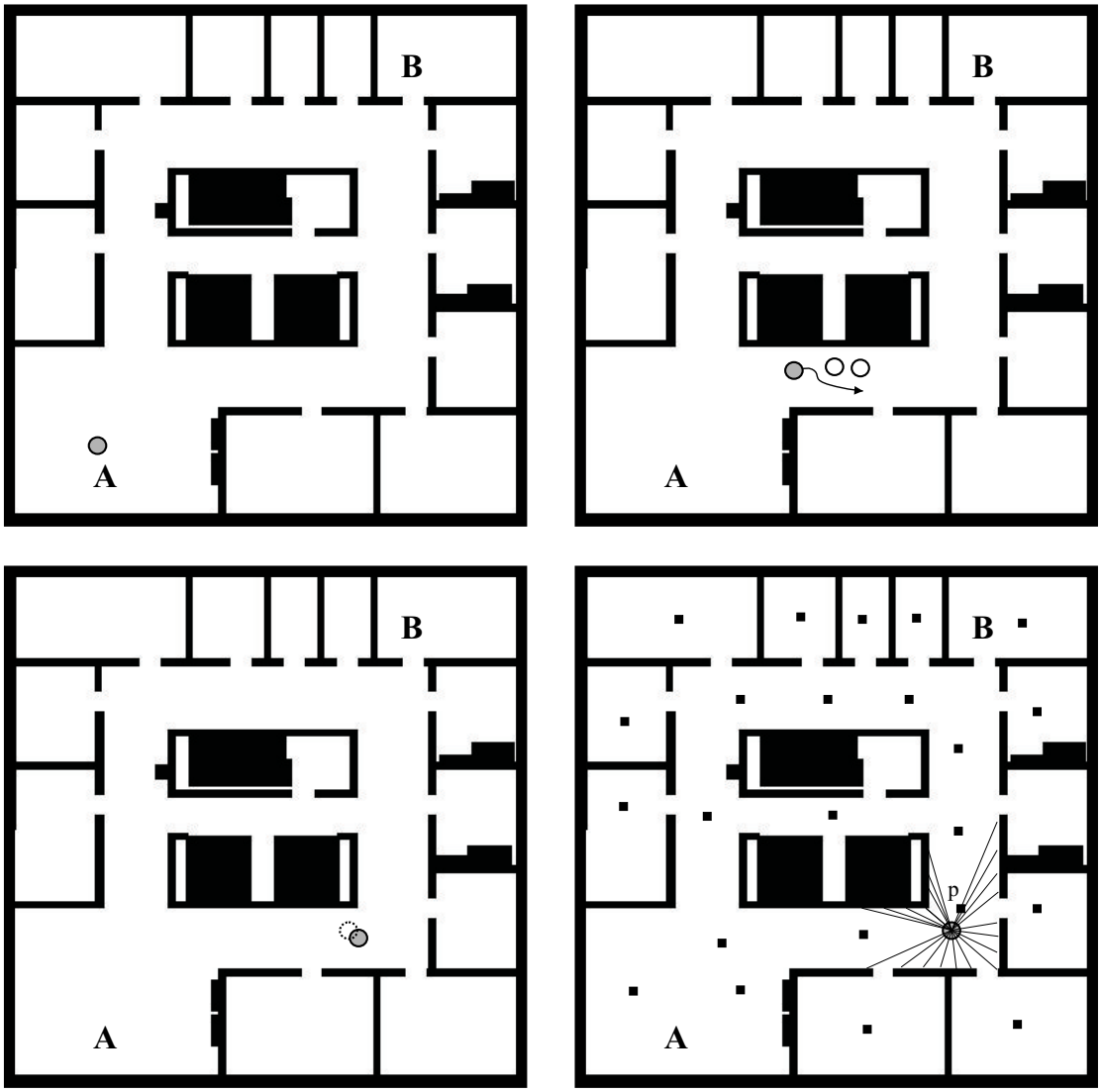


Fig. 3. A schematic illustration of a sequence of typical situations encountered by the transportation robot. See the main text for a more detailed description of the four panels in this figure.

The battery of the robot should be such as to allow continuous operation for several hours, and preferably for a full working day. This might be difficult to achieve and the robot should therefore be equipped with a *battery charging* behavior (B6), allowing it to locate a charging station and charge its batteries when needed. However, for simplicity, the first prototype of the robot will not be equipped with such a behavior. Thus, the problem of battery charging will not be considered further in this paper. Even in the absence of battery charging, the development of a robot capable of carrying out the task outlined above will be a challenging task. Here, a few details concerning some of the challenges will be given.

B. Challenges

1) Safety: First and foremost, the robot must operate in a safe way, i.e. it must never collide with people or obstacles. Note that, even though the robot is equipped with a map, it may still encounter stationary obstacles. An example is the case of a hospital environment, where many different stationary (but movable) objects may be present in certain situations, and absent in others.

In an encounter with a single person, avoiding collisions is not so difficult. However, in a congested environment, the problem becomes more difficult. For example, if the robot moves backwards quickly in order to avoid a collision, it may bump into a person behind the robot. The robot's first action, therefore, will always be to stop if an obstacle is detected in the direction of motion. This will have the additional benefit of making the robot's behavior predictable from the point-of-view of the people working in the same environment.

Another possibility will be for the robot to choose a different way, in case its current path (as obtained from the A* algorithm) is blocked. Here, however, the robot must be careful not to change its path too frequently, as this may result in a considerable delay in the delivery of the

transported object.

2) Snapshot matching: As is well known, reliable self-localization is a common difficulty encountered in navigation problems involving autonomous robots [18], [5].

In order to recalibrate its odometry, the robot developed in this project must find and match its current location against stored snapshots. Clearly, other options exist for localization, such as e.g. the NorthStar system [12]. However, this project is aimed at achieving navigation without any adaptation of the environment, such as installation of beacons, transmitters, or other hardware [9]. In addition, the snapshot matching method has a biological equivalent in the procedure used by some species of ants [8], [3], [4], and is interesting in its own right, particularly in the light of the biologically inspired approach to behavior selection defined by the utility function method. The snapshot matching could, of course, also be based on vision using two video cameras, and the use of binocular vision is certainly retained as a possibility. However, the simulations carried out so far have indicated that the 2D laser range finder ought to be sufficient for the snapshot matching, provided that it is carried out frequently.

3) Sensory integration: In order for the robot to operate robustly, it should preferably be able to navigate even if some sensory modality fails. For example, if the IR proximity sensors suddenly break, the robot should be able to switch to alternative proximity detection methods, e.g. based on the laser range finder readings. This would not be optimal, since the range finder will be located at a different height than the proximity sensors, and may therefore miss certain obstacles that would have been detected by the IR sensors. A possible solution, in case of IR sensor failure, is to navigate more slowly, using a combination of the readings from the laser range finder and the bumper sensors. An alternative approach is to provide the robot with sensor redundancy, using e.g. two sets of IR proximity detectors, or a sonar. The problem of sensor failure can thus be solved either mainly as a software problem (dynamically switching from IR sensors to the laser range finder in case the former break down) or mainly as a hardware problem (providing the robot with redundant sensors).

4) Behavior selection: From a software point of view, one of the main challenges is behavior selection. The problem is made more difficult by the fact that the robot will operate in an unstructured, rapidly changing environment. Clearly, the robot must always avoid collisions with people (see Subsect. II-B.1 above) or stationary obstacles, but it will nevertheless operate under conditions that require a certain trade-off: If the robot is made *too* careful, it will most likely move too slowly to be useful. A similar problem will occur if, for example, the robot misjudges the amount of congestion along a certain path and unnecessarily selects a much longer path. Thus, finding the right balance between efficiency on the one hand, and safety and self-preservation on the other, is likely to be one of the main challenges encountered during the evolution of the behavior selection system.

Another, related, challenge is to evolve a behavior selection system that is sufficiently general, so that it can cope with any situation (within reasonable limits) that may occur. In view of the rather long time taken to evaluate robots in simulations, this problem will be far from trivial.

III. The utility function method

Behavior selection (also called behavioral organization or action selection), i.e. the problem of activating (in any situation) the correct behavior among the behaviors available in a robot's behavioral repertoire, is a challenging task that has been approached in many different ways (see e.g. [14] for a review).

The utility function (UF) method [22], [23], [21] is a method for behavior selection based on evolutionary optimization of utility functions. It is described in detail by Wahde [22] and therefore only a brief introduction will be given here.

A. Brief description

The UF method is an arbitration method, i.e. a behavior selection method in which a single behavior is active at any given time. The method deals with the *selection* of behaviors that are already present. Thus, in order to apply the method, one must first generate a set of basic behaviors (e.g. by hand, in simple cases, or using evolutionary optimization in more complex cases). Some examples of behaviors are described in the project outline above.

In the UF method, a set of state variables is defined. These can be of three kinds: (1) External variables (denoted $\mathbf{s}$) based e.g. on the readings of IR proximity sensors or a laser range finder, (2) internal physical variables (denoted $\mathbf{p}$) measuring e.g. the energy level in the robot's batteries and (3) internal abstract variables (denoted $\mathbf{x}$), whose variation may be either hand-coded or evolved, and which roughly correspond to (the action of) hormones in biological systems. For example, an internal abstract variable can be used to model fear. In that case, its value would rise e.g. in cases where a collision or battery depletion is imminent.

Each behavior B_i contained in the brain of the robot is associated with a utility function U_i that depends on (a subset of) the state variables, i.e.

$$U_i = U_i(\mathbf{s}, \mathbf{p}, \mathbf{x}), \quad i = 1, \ldots, n, \tag{1}$$

where n is the number of behaviors available.

Once the utility functions have been generated, behavior selection is straightforward: At any given time, the robot simply activates the behavior corresponding to the largest utility value, i.e.

$$i_{\text{active}} = \text{argmax}\left(U_i\right), \tag{2}$$

where i_{active} denotes the index of the currently active behavior. Thus, in this method, the utility values are used as a common currency [22], [11] allowing the robot to assess, on a dynamical basis, the relative merit of the different behaviors.

The problem, of course, is to generate the utility functions. In the UF method [22], the utility functions are optimized by means of an evolutionary algorithm. This procedure is carried

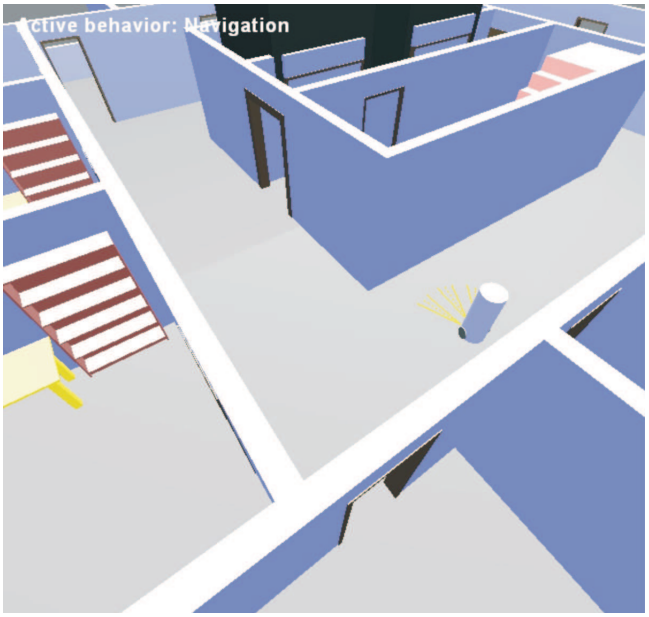


Fig. 4. A snapshot from a simulation based on the UFLib simulation package.

out in simulations, based on the UFLib software package, which will now be described briefly.

B. Software

The application of the UF method requires that many different behavior selection systems (i.e. sets of utility functions) should be evaluated. In order for this to be possible, simulations must normally be used. The UF method has been implemented in the UFLib software package [20]. Written in Delphi (object-oriented Pascal) the UFlib package contains software for 3D simulation of wheeled robots in arbitrary arenas, using the UF method for behavior selection. The package also implements an evolutionary algorithm allowing evolution of the utility functions that determine the behavior selection. UFLib supports multiple evaluations, so that each behavior selection system can be tested in a variety of situations. Furthermore, the software package supports the use of behavioral hierarchies, i.e. layers of sub-behaviors within each behavior. However, these concepts will not be described further in this paper. Note that the current version of UFLib can be downloaded for academic use [20].

IV. Preliminary results

Until the present time (June 2006), the main work in the project has been the development of the necessary software. A significant amount of time has been spent on completing the UFLib [20], and testing it in various circumstances [21], [23]. Furthermore, all of the required behaviors (B1 - B5 as listed above), except B3, have been completed. In particular, the *navigation* behavior B1 has been finalized and thoroughly tested in simulation. A snapshot from such a test is shown in Fig. 4.

Recently, a specification of the hardware requirements has been made, and the initial design phase has been started.

A. Future work

The next step is to complete the hardware design, and then to begin hardware construction. Obviously, this will be an iterative process, involving both system identification aimed at making the simulator as accurate as possible, and repeated modification of both hardware and software.

The aim is to have a working prototype completed in the spring of 2007.

Acknowledgements

The authors would like thank Mr. Krister Wolff (Chalmers Univ. of Technology), Drs. Pitoyo Hartono (Future Univ., Hakodate) Kenji Suzuki (Tsukuba Univ.), Ryo Saegusa (Waseda Univ.), and Prof. Shuji Hashimoto (Waseda Univ.) for helpful discussions during the preparation of this paper. Also, the project participants wish to thank the Carl Trygger foundation for its financial support for this project.

References

[1] AIBO, (Sony, www.sony.net/Products/aibo).
[2] R. Arkin, *Behavior-based robotics*. MIT Press, 1998.
[3] T. S. Collett and M. Collett, "Memory use in insect navigation," *Nature Reviews Neuroscience*, vol. 3, pp. 542–552, 2002.
[4] M. O. Franz, B. Schöllkopf, H. A. Mallot, and H. H. Bülthoff, "Where did i take that snapshot? scene-based homing by image matching," *Biological Cybernetics*, vol. 79, pp. 191–202, 1998.
[5] U. Frese, "A discussion of simultaneous localization and mapping," *Autonomous Robots*, vol. 20, no. 1, pp. 25–42, 2006.
[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths in graphs," *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, July 1968.
[7] HRP 2, (Kawada Industries www.kawada.co.jp/global/ams).
[8] S. P. D. Judd and T. S. Collett, "Multiple stored views and landmark guidance in ants," *Nature*, vol. 392, pp. 710–714, 1998.
[9] C. Kee *et al.*, "Development of indoor navigation system using asynchronous pseudolites," in *Proceedings of ION GPS-2000*, 2000, pp. 1038–1045.
[10] MB385, (BlueBotics, www.bluebotics.com/automation/MB835/).
[11] D. McFarland and T. Bösser, *Intelligent behavior in animals and robots*. Cambridge, MA, USA: MIT Press, 1993.
[12] Northstar (Evolution robotics, www.evolution.com).
[13] Papero (NEC, www.incx.nec.co.jp/robot).
[14] P. Pirjanian, "Behavior coordination mechanisms – state-of-the-art," Institute of Robotics and Intelligent Systems, USC, Los Angeles, Tech. Rep., 1999.
[15] M. E. Pollack *et al.*, "Pearl: A mobile robotic assistant for the elderly," in *AAAI Workshop on Automation as Caregiver*, August 2002.
[16] Roomba (iRobot, www.irobot.com).
[17] R. Simmons, R. Goodwin, K. Z. Haigh, S. Koenig, and J. O'Sullivan, "A layered architecture for office delivery robots," in *Proceedings of the first international conference on Autonomous agents*, 1997, pp. 245–252.
[18] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. The MIT Press, 2005.
[19] TUG (University of Maryland, www.umm.edu/news/releases/robot.html).
[20] www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary.
[21] M. Wahde, J. Pettersson, H. Sandholt, and K. Wolff, "Behavioral selection using the utility function method: A case study involving a simple guard robot," in *Proc. of the* 3rd *Int. Symp. on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, 2005, pp. 261–266.
[22] M. Wahde, "A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions," *Journal of Systems and Control Engineering*, vol. 217, pp. 249–258, 2003.
[23] M. Wahde and J. Pettersson, "Application of the utility function method for behavioral organization in a locomotion task," *IEEE Trans. Evol. Comp.*, vol. 9, no. 5, pp. 506–521, 2005.

Paper X

Behavior selection for localization and navigation in a transportation robot, using evolvable internal state variables

manuscript to be submitted to

Autonomous Robots

Behavior selection for localization and navigation in a transportation robot, using evolvable internal state variables

Jimmy Pettersson and Mattias Wahde

Chalmers University of Technology
Department of Applied Mechanics
412 96 Göteborg, Sweden
E-mail: {jimmy.pettersson, mattias.wahde}@chalmers.se

Abstract The utility function (UF) method for behavior selection has been applied to the problem of providing a (simulated) robot with the means to navigate accurately between two arbitrarily chosen points A and B in a given arena, while avoiding collisions and keeping track of its location.

The UF method is an arbitration method based on evolutionary optimization of utility functions, and one of its strengths is that it reduces (compared to most methods of behavior selection) the number of parameters that must be hand-tuned by the user. However, in earlier work on the UF method some hand-tuning has remained, notably in the dynamics of the internal abstract variables of the robotic brain (also referred to as hormone variables). Here, the UF method has been extended so that both the utility functions and the dynamics of the hormone variables now are obtained through evolutionary optimization.

Key words Behavior-based robotics, behavioral organization, utility functions, localization

---

1 Introduction and motivation

One of the main difficulties in the behavior-based approach to robotics is behavioral organization (behavior selection), i.e. the problem of activating an appropriate behavior at all times [1,2]. Several methods have been proposed for solving this problem, but most methods are either intended for a particular application or rely heavily on the ability of the experimenter to set the values of numerous parameters by hand [3–7].

The behavior-based approach has its origin in ethological considerations and, recently, methods for behavior selection inspired by their biological counterparts have been considered [8,9]. One such approach is the utility function (UF) method [9], where the concept of utility [10–12] is used as a common currency in robotic decision-making. In the UF method, the number of parameters that must be set by hand is greatly reduced, since the utility functions (that determine the selection of behaviors) are obtained by means of an evolutionary algorithm (EA).

However, in addition to the utility functions, the UF method also relies on so called *internal abstract variables* (also called *hormone variables*) that roughly correspond to signalling molecules (such as hormones) in biological systems and the variation of these variables have, until now, been specified by hand. Indeed, in the studies concerning the UF method reported in several recent papers [13–15] it has been quite easy to do so, since those studies were aimed at investigating the basic properties of the method, and the problems were thus kept quite simple. In this paper, the UF method will be applied to a considerably more complex problem (see below) in which the variation of the hormone variables is much more difficult to set by hand. Thus, in order to retain the claim of reducing the amount of parameter hand-tuning, the UF method should be extended to allow for evolutionary optimization of the variation of such variables.

An additional aim of this paper is to make full use of the UF method in a more complex and realistic application, namely the first steps of the development of a general-purpose robot for transportation and delivery. Among other things, such a robot must be capable of reliable and collision-free navigation in unpredictable indoor environments (where GPS cannot easily be used [16]), without modification of the environment. The goal of this research project, which involves researchers at several universities in Sweden and Japan, is to generate a delivery robot using the UF method for behavior selection. The project is also described in [17].

2 Problem description

The intended operation of the transportation robot is best described through an example. Consider the envi-

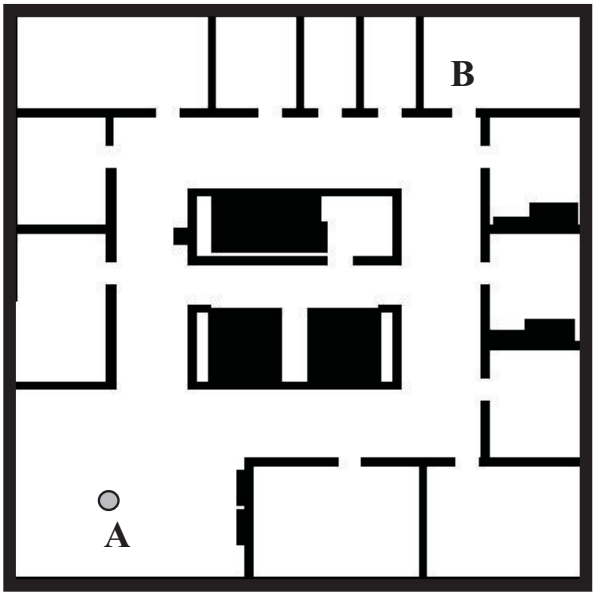


Fig. 1 A schematic illustration of the transportation robot in a typical arena. The task of the robot, shown as a disk near the lower left corner of the arena, is to transport objects from point A to point B (arbitrarily chosen).

ronment (also referred to as the *arena*) shown in Fig. 1. The robot starts at some arbitrary point A. The user will place an object to be transported on the robot and instruct it via a user interface (e.g. a voice command) to go to a specific point B (the location of which may, of course, vary from case to case). The robot is assumed to be equipped with a map of the arena. Based on the map, the robot will plot a path to B (details will be given below), and begin executing the motion. In order to keep track of its position, the differentially steered robot will be equipped with two sensory modalities, namely odometry based on encoder readings from each wheel and positioning (localization) based on the readings of a laser range finder (LRF) mounted on the robot.

As the robot moves, the position estimates obtained from the odometry will drift and, as navigation depends on accurate positioning, calibration will therefore be necessary at regular intervals. For calibration, the robot will use a compass to adjust its heading, and then carry out a procedure during with the current readings of its LRF are matched to those obtained using a simulated range finder taking virtual measurements based on the map. Once a sufficiently close match has been found, the robot can localize itself, i.e. calibrate its estimated position and heading, and then resume its motion. When point B is reached, the robot will stop and await further orders.

Of course, in an office environment, the robot is likely to encounter moving obstacles (i.e. people) along the path from point A to point B. Thus, the robot should also be equipped with the ability to avoid obstacles. Obstacle avoidance can be triggered e.g. by the readings of IR proximity detectors placed around the body of the robot. However, in the first step of this project, presented in this paper, moving obstacles will not be considered. Nevertheless, obstacle avoidance will be needed even in a completely static environment; this is so, since the drift in the odometry and the possibility of slightly incorrect position estimates generated during localization, may sometimes lead to collisions with the walls of the arena (or with other stationary objects). In such cases, touch sensors protruding from the body of the robot can be used for indicating whether or not the robot touches an obstacle.

The robot will thus need (at least) three behaviors: *path navigation*[1], *localization*, and *obstacle avoidance*. These behaviors will be described in detail in Sect. 5.

While the task described above may sound simple, the construction of a robot capable of reaching the stated objectives involves numerous difficulties. First of all, accurate localization, which has been considered by many authors (see e.g. [18–21]) is a notoriously difficult problem. With the sensory modalities used in the robot described above, some of the difficulties are (1) rapidly and accurately matching current readings to virtual readings during navigation, (2) deciding when to recalibrate the odometric readings during navigation, (3) making sure to avoid collisions with obstacles, moving or stationary, and (4) keeping the odometry reasonably accurate during (possibly rapid) swerving while avoiding obstacles. Furthermore, in a commercially viable robot, the time wasted on obstacle avoidance and localization should be kept to a minimum. Thus, the robot must wait as long as possible (but not longer!) before activating either of these two behaviors. On the other hand, the robot must, of course, actually *reach* the intended goal to be useful at all.

Put in a different way, the problem of behavioral selection (behavioral organization) is of paramount importance for this robot and the procedure of generating the brain of the robot can be formulated as an optimization problem: For any points {A,B}, minimize the time taken for the motion from A to B, subject to the constraint that the robot should not collide with any obstacles (and the implicit constraint that it should, at all times, know its current location and heading). In this paper, the behavior selection system will be generated by means of the utility function (UF) method [9], which will now be described.

3 The utility function method

The UF method and its basic properties have been thoroughly studied and described in previous papers [13–15] and therefore only a brief description of the method will be given here. However, the new feature added to the UF method, namely the evolution of the dynamics of hormone variables, will be described in detail.

[1] Names of behaviors are typeset in *italics*.

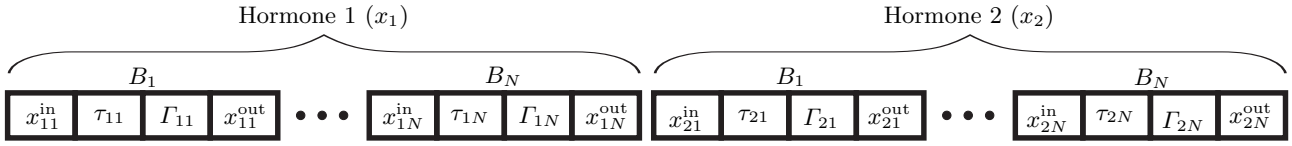


Fig. 2 A schematic illustration of the chromosome determining the dynamics of the hormone variables. For each of the M hormones ($M = 2$ in the figure), the chromosome contains one gene (encoding the parameters x^{in}, τ, Γ, and x^{out}) for each of the N behaviors. Thus, the total number of genes used for specifying the $4N \times M$ parameters of the hormone dynamics equals $N \times M$. The four parameters in each gene determine the modification (if any) of the hormone level upon entry to the behavior in question, the variation of the hormone level as long as the behavior remains active, and the modification (if any) of the hormone level upon exit from the behavior; see the main text for a detailed description.

3.1 Basic description

The UF method is an arbitration method, i.e. a method in which only one behavior is active at any given time. In the UF method, each behavior is associated with a utility function, whose value depends on the state of the robot, which, in turn, is given by the state variables $\mathbf{z} = \{\mathbf{s}, \mathbf{p}, \mathbf{x}\}$, where $\mathbf{s}$ are external variables (e.g. the readings of IR sensors), $\mathbf{p}$ internal physical variables (such as e.g. the energy level in the robot's battery) and $\mathbf{x}$ are internal abstract (hormone) variables. The internal variables (physical and abstract) determine the internal state of the robot. However, whereas the physical variables are, as their name implies, obtained through measurements of physical quantities, the abstract variables (roughly) represent signalling molecules in biological systems, and it is these variables that introduce a non-reactive component in the robotic brain, i.e. they provide it with what one might call a rudimentary artificial endocrine system[2]. This system, in turn, acts essentially as a short-term memory. For example, an increase in a hormone variable x in a given situation will alter the state of the robot and, depending on the dynamics of the variable (e.g. its decay rate), the robot might take a different action should it (shortly thereafter) find itself in the same situation again.

Mathematically, the utility function for behavior B_i is specified as $U_i = U_i(\mathbf{s}, \mathbf{p}, \mathbf{x})$ and, commonly, a polynomial ansatz (with a given polynomial degree d) is used for each utility function. As an example, the ansatz for a utility function $U_i = U_i(s, p)$ of two variables, and with polynomial degree $d = 2$, will take the form

$$U_i(s,p) = a_{00}^{(i)} + a_{10}^{(i)} s + a_{01}^{(i)} p + a_{20}^{(i)} s^2 + a_{11}^{(i)} sp + a_{02}^{(i)} p^2, \quad (1)$$

where the $a_{ij}^{(i)}$ are constants to be determined (see below). Note that not all utility functions must depend on all state variables; in many cases, the utility functions depend only on a subset of the available state variables.

Behavior selection is simple in the UF method: At any given time, the behavior with the highest utility (as obtained from its utility function) is active. The problem, of course, is to determine the utility functions, i.e. to determine the constants in all utility function polynomials. In the UF method these functions are obtained through evolutionary optimization. Thus, application of the UF method (normally) requires the use of simulations. A software library (UFLib) implementing the UF method (including also robot dynamics and 3D visualization) has been described in earlier work [22–24], and will also be used in the simulations carried out in this paper.

3.2 Dynamics of internal abstract (hormone) variables

As mentioned in the introduction, in earlier work, the dynamics of the hormone variables was specified by hand. In this paper, the first steps will be taken towards allowing evolutionary optimization not only of the utility functions but also of the hormone dynamics.

The variation of each hormone variable x_i in a given behavior j is determined by the four parameters x_{ij}^{in}, Γ_{ij}, τ_{ij}, and x_{ij}^{out}, which are encoded in the genome, as illustrated in Fig 2. The parameters are used as follows: Upon entry to behavior j, x_i is set according to

$$x_i = x_{ij}^{\text{in}}. \quad (2)$$

Similarly, upon exit from the behavior, x_i is set as

$$x_i = x_{ij}^{\text{out}}. \quad (3)$$

Thus, the value of x_i normally varies discontinuously when a behavior switch occurs. During the active period of behavior j, i.e. after entry but before exit, x_i varies according to

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = \frac{\Gamma_{ij} x_i^{\max} - x_i}{\tau_{ij}}, \quad (4)$$

where $x_i^{\max}$ is the maximum allowed level of x_i (usually set to 1). Γ_{ij} is an integer parameter taking the value 0 or 1. Thus x_i rises exponentially towards the maximum level if $\Gamma_{ij} = 1$, and falls off exponentially towards the minimum level (=0) if $\Gamma_{ij} = 0$.

Note that the implementation described above is just one among many possible options for the x_i, in which the variation only depends on time (except for the discontinuous jumps at entry and exit). In principle, x_i could,

[2] Note, however, that the dynamics of the hormone variables might, in some cases, be much faster than the dynamics of hormones in biological systems.

of course, depend also on e.g. sensor readings and on the values of the other hormone variables. However, the present implementation has the advantage of low complexity (only 4 parameters per behavior, for each x_i) and certainly represents a great improvement in flexibility over the procedure used earlier, in which the x_i were normally set (by hand) only at entry and exit. Note, however, that this particular type of variation remains as an asymptotic special case, obtained if the EA chooses to set the τ_{ij} to very large values.

3.3 Parameters of the behavior selection system

With the addition of the hormone dynamics just described, the behavior selection system in the UF method will depend on (1) the coefficients of the utility function polynomials and (2) the parameters determining the hormone dynamics. It can be shown that the number of terms N_t, and therefore the number of coefficients, in a polynomial of n variables and degree d equals

$$N_t = \binom{n+d}{n} \equiv \binom{n+d}{d}. \quad (5)$$

Thus, in a behavior selection problem involving M hormones and N behaviors, each associated with a utility function polynomial of n_i state variables ($i = 1, \ldots, N$) and degree d, the total number of parameters that must be specified equals

$$N_p = 4N \times M + \sum_{i=1}^{N} \binom{n_i+d}{d}. \quad (6)$$

Thus, the number of parameters will depend on the number of behaviors, the number of hormones, the number of state variables in each behavior, and the polynomial degree.

4 Simulation setup

In order to evolve the behavior selection system, i.e. to set the values of its N_p parameters, simulations were carried out. Before simulations can be started, however, it is necessary to specify (1) the arena, i.e. the placement of walls and stationary objects such as bookshelves, desks etc., (2) the body of the robot, i.e. its size and shape, the position of its sensors etc., and (3) the behavioral repertoire of the robotic brain. Once the arena, the robot body, and the behavioral repertoire have been specified, simulations aimed at optimizing the behavior selection system (including the hormone dynamics) can begin. For a detailed description of the procedure needed for setting up a simulation based on UFLib, see [22].

This section begins with a description of the arena and the robot (body and brain) used in the simulations. Next, the evolutionary procedure for optimizing the behavior selection system is described.

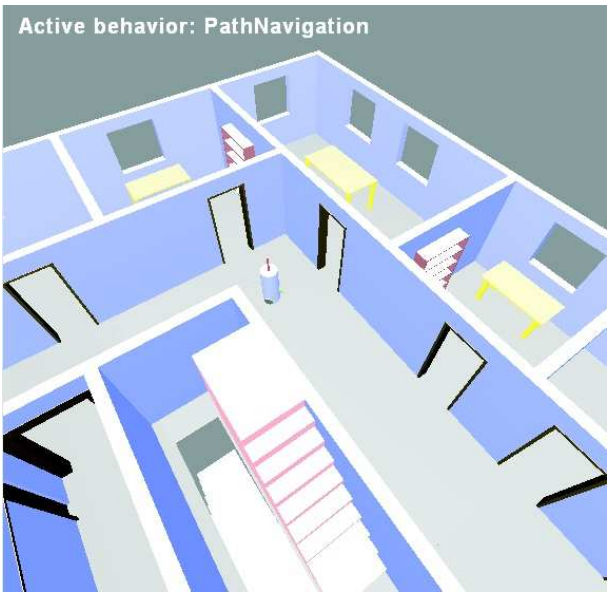


Fig. 3 A three-dimensional representation of a part of the arena, generated by the visualization routines contained in UFLib. In the snapshot shown here, the robot is seen navigating along a corridor.

4.1 Arena

The simulations were carried out in an arena corresponding to a typical office environment, shown from above in Fig. 1 and in 3D in Fig. 3. The center of the arena contains a staircase and elevators. These two regions were off-limits for the robot. Thus, the robot was constrained to move in the corridors and offices.

4.2 Robot body

A differentially steered robot with cylindrical cross section was used. The weight of the robot was 10.0 kg, its radius and height were 0.20 m and 1.0 m, respectively, and the wheel radius was 0.10 m. The robot was equipped with a laser range finder (LRF), mounted on a pole attached to the cylindrical body of the robot. The simulation model of the LRF was based on the Hokuyo URG-04LX range finder. The robot was also fitted with two encoders (one for each wheel) for odometry, a compass, and three touch sensors protruding from the front of the robot in the directions -60°, 0°, and 60° relative to the forward direction. Since moving obstacles were not considered, there was no need to equip the robot with IR proximity detectors (see also Sect. 2 above), even though such sensors could easily have been added. An illustration of the robot is shown in Fig. 4.

All real sensors are, of course, noisy, and UFLib therefore supports the addition of noise at all relevant levels. Thus, during simulations based on UFLib, noise is added both to motor torques and sensor readings. The noise level used for the simulated LRF was based on actual

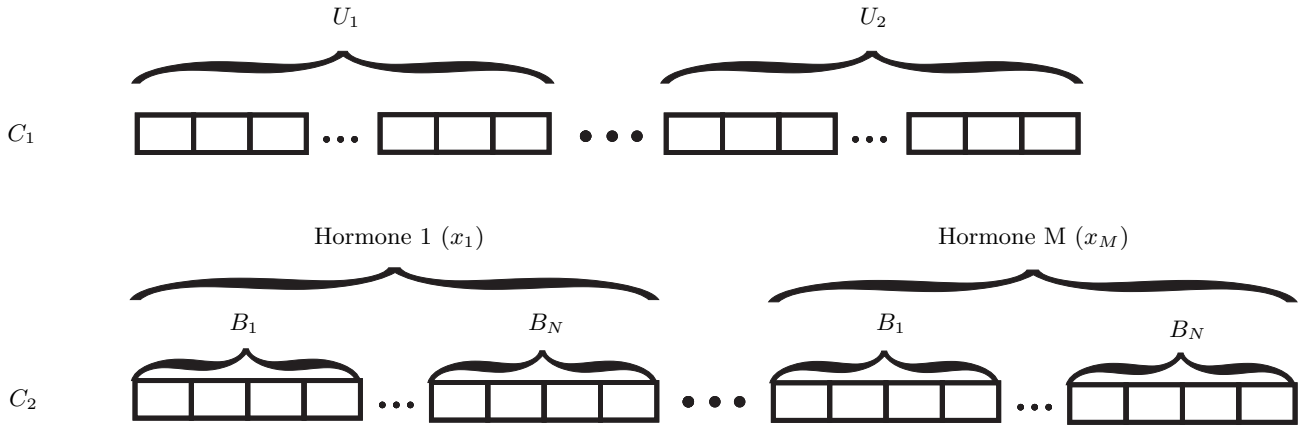


Fig. 5 A schematic illustration of the two chromosomes determining the utility functions and the hormone dynamics. The upper chromsome (C_1) encodes the utility functions. Each box in C_1 represents the coefficient for one polynomial term; see also Eq. (1). The lower chromosome (C_2) encodes the hormone dynamics. Here, each set of four boxes represents the parameters x^{in}, τ, Γ, and x^{out} that determine the variation of one hormone variable during periods of activity of one particular behavior, as illustrated in Fig. 2.

Sensor	Readings	Accuracy
Laser range finder	682 rays, 240° sector, range 0.0-4.0m	±0.001 m
Compass	0-360°	$\pm \sim 3°$
Wheel encoder	1024 pulses/revolution	$\pm \sim 2$ pulses per time step
Touch sensor	0 or 1	–

Table 1 Ranges of measurement and noise levels for the sensors used in the simulated robot.

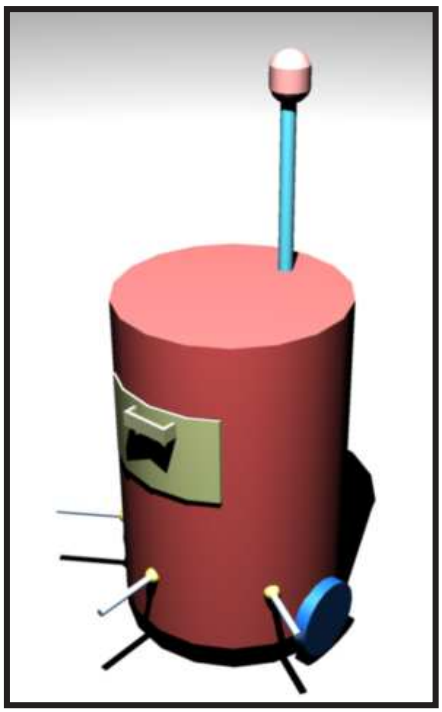


Fig. 4 The transportation robot used in the simulations. The laser range finder is mounted at the top of the pole.

readings from an Hokuyo URG-04LX laser range finder. The noise levels for the other sensors were estimates, based on typical values from manufacturers' data sheets. A description of the various sensors is given in Table 1. Note that with 1024 pulses per revolution of the wheels (radius: 0.10 m), and a typical robot speed of around 0.7 m/s, an average of around 11 pulses are detected, per time step[3], by the wheel encoders. The error in the detection is up to ± 2 pulses *per time step*. Thus, the odometric drift can be quite significant.

4.3 Robot brain

The specification of a robotic brain in the UF method requires (1) a behavioral repertoire, (2) the utility functions used for selecting between behaviors, and (3) equations determining the dynamics of the hormone variables.

In the experiments considered here, three behaviors were included in the behavioral repertoire, namely *path navigation* (B_1), and *localization* (B_2), and *obstacle avoidance* (B_3). The behaviors are described in detail in Sect. 5 below.

While the user must provide the behavioral repertoire, the UF method itself generates the behavior selection system (the utility functions) and the parameters for the dynamics of the hormone variables. However, the user must, of course, specify which state variables are to be used in each behavior. In this investigation, two hormone variables x_1 and x_2 were introduced, and these

[3] The length of the time step was 0.01 s.

were the only state variables used for B_1 and B_2. For B_3, the readings of the three touch sensors were used as state variables (s_1, s_2, and s_3), together with the two hormone variables. Thus, the utility function polynomials were specified as

$$U_1 = U_1(x_1, x_2) = a_{00}^{(1)} + a_{10}^{(1)} x_1 + \ldots, \quad (7)$$

$$U_2 = U_2(x_1, x_2) = a_{00}^{(2)} + a_{10}^{(2)} x_1 + \ldots, \quad (8)$$

and

$$U_3 = U_3(x_1, x_2, s_1, s_2, s_3) = a_{00000}^{(3)} + a_{10000}^{(3)} x_1 + \ldots, \quad (9)$$

for B_1, B_2, and B_3, respectively. Following the results obtained in earlier studies [13], the polynomial degree was set to 3. Thus, with $N = 3$, $M = 2$, $n_1 = 2$, $n_2 = 2$, $n_3 = 5$, and $d = 3$, the total number of parameters, obtained from Eq. (6), was 100.

4.4 Optimization procedure

As mentioned above, the simulation program was based on the UFLib software library, which was expanded to include evolutionary optimization of the hormone dynamics. Each individual in the EA contained a behavior selection system encoded in two chromosomes, determining the N_p parameters, as illustrated in Fig. 5. Thus, in the decoding step, the two chromosomes of an individual generated a complete behavior selection system, which was then evaluated by letting the corresponding robot move in the arena shown in Fig. 1. A more complete description of the evaluations is given in Sect. 6. The fitness measure, which is described in Subsect. 6.2 was based on the navigation performance of the robot.

A fairly standard EA (implemented in UFLib) was used, with generational replacement (after the evaluation of all individuals), based on tournament selection, crossover, and parametric mutation. Note that the crossover operator may only cut chromosomes *between* genes. In UFLib, the utility functions encoded in the first type of chromosome (C_1) in Fig. 5, are each represented by *one* gene. Thus, for N behaviors, there are $N - 1$ possible crossover points available when crossing two chromosomes of that type. Similarly, in the second type of chromosome (C_2, in Fig. 5), each set of four parameters x^{in}, τ, Γ, and x^{out} is represented by *one* gene, and the number of crossover points therefore equals $N \times M - 1$ (see also Fig. 2). In the formation of two new individuals, genetic material is exchanged between the two parent individuals on a chromosome-by-chromosome basis, i.e. by first crossing the C_1 chromosomes of each parent, and then the C_2 chromosomes.

Mutations are carried out parameter by parameter in UFlib. Since the total number of parameters was equal to 100 (see Subsect. 4.3), the mutation rate was set to $p_{\text{mut}} = 0.03$, following the suggestion of using $p_{\text{mut}} \approx 3/N_p$ given in [25]. The population size was set to 30, and the crossover probability p_{cross} to 0.50. The tournament selection parameter, i.e. the probability of selecting the better of the two individuals in a tournament was equal to 0.70.

5 Behaviors

5.1 Path navigation

In this behavior the robot navigates through a sequence of waypoints, generated through the combined use of a grid-based map and an A* search algorithm [26,27], which is capable of quickly generating the shortest path between two locations in the map. Once a path has been generated, the operation of the path navigation is as follows: The first waypoint from the sequence of generated waypoints is retrieved and the robot starts to navigate towards the location of the waypoint. The navigation is performed by setting the motor commands in such a way that the robot travels in one of three ways: (1) turning left, (2) turning right, or (3) going forward. If the distance to the current waypoint is below a certain threshold (set by the user), the next waypoint is retrieved from the list. Each waypoint is processed in a similar fashion until the last one is retrieved. The last waypoint coincides with the target location at which the robot reaches a standstill by setting the motor signals to zero.

Whenever the robot reaches a target, a new target location is generated (and thus a new sequence of waypoints). Depending on the particular settings used, the new target location is either taken from a list specified by the user, or generated randomly in such a way that the distance from the robot's position to the new target location is uniformly distributed in the range $[d_{\min}, d_{\max}]$. In this way, the average length of the paths is kept close to constant between different simulations, thus making possible a fair comparison of solution candidates.

If the behavior selection mechanism temporarily activates a different behavior (e.g. *obstacle avoidance*), the path towards the current target is regenerated upon reentry to the *path navigation* behavior, and a new sequence of waypoints is obtained.

It should be noted that the *path navigation* behavior relies solely on odometry. Consequently, in order to navigate successfully to a designated target, the estimated position of the robot must be sufficiently accurate in order to avoid large deviations from the intended path, which may lead to collisions with obstacles.

5.2 Localization

The *localization* behavior used here is based on the readings of the LRF which is mounted on a pole extending vertically from the top surface of the robot, to avoid including moving objects (if any) in the scan. The basic idea is to match the current readings of the actual LRF

to the readings of a virtual LRF placed (virtually) in various locations in the map, in the vicinity of the estimated position of the robot[4]. The *localization* behavior works as follows: (1) Upon activation of the behavior, the robot first checks (using the wheel encoders) whether or not it is moving. If it is, it begins by braking until it reaches a complete standstill, a process that usually takes around $T_s = 0.10 - 0.40$ s (10-40 time steps in the simulation) to complete. Next, (2) the robot scans the environment using its laser range finder (LRF). Obtaining the reading of a Hokuyo LRF takes around $T_{\mathrm{LRF}} = 0.10$ s (10 time steps), and this is accounted for in the simulation, in order to make it as realistic as possible. In this 0.10 s interval, the LRF is able to measure the distance to the nearest obstacle along at total of 682 directions (henceforth referred to as *rays*) distributed evenly in a 240 degree sector. Since the readings of any sensor always contain some noise, an average of three readings is taken. Next, (3) the robot begins matching the stored readings from the actual LRF to virtual readings generated using the onboard map of the environment. During this part of the localization behavior, the robot begins by calibrating its direction to the readings of the onboard compass[5]. The robot then places a virtual LRF in various locations in the map (stored in its brain), generates (based on the map) three virtual LRF readings (using the same noise level as in the actual LRF), stores the results in a vector and, finally, compares the virtual readings to the actual readings obtained in step 2. The virtual LRF is an artificial construct contained in the brain of the robot (i.e. it does not correspond to a hardware component), and it can therefore be read much faster than the actual simulated LRF used in step 2. Thus, in a single time step of the simulation (0.01 s), the robot completes the three virtual readings for *one* given position of the virtual LRF, as well as the matching of the reading thus obtained to the stored reading of the actual LRF.

The procedure for placing the virtual LRF in a sequence of positions in the map can be chosen in many different ways. Here, it is done as follows: First, a virtual reading is taken at the robot's estimated position $\hat{\mathbf{x}}$. In the absence of odometric drift (and sensory noise) this reading would be identical to the reading from the actual LRF, and the estimated position would be identical to the actual position $\mathbf{x}$. However, due to the odometric drift and noise, the two readings will of course differ. Thus, additional virtual readings are taken (one per time step) along concentric circles (see Fig. 6), centered on the robot's estimated position. The sequence of positions $\mathbf{p}_k$ is parametrized by Δ_r, measuring the radial steps (i.e. the distance between the concentric circles) and Δ_ϕ, measuring the angle between successive positions on a given circle. Thus, the number of tested positions per circle equals

$$N_c = 2\pi/\Delta_\phi, \tag{10}$$

assuming that Δ_ϕ has been set so that N_c is an integer. When a particular circle has been completed, i.e. when all the N_c positions have been tested, the radius is increased by Δ_r.

Note that, in each position $\mathbf{p}_k$, the robot could, in principle, rotate the virtual LRF to generate virtual readings also for different *directions*. However, as mentioned above, the direction is instead obtained from the robot's compass, and the matching procedure in step 3 of the localization behavior thus only concerns the *position* of the robot.

Thus, in step 3, for each time step, a new position $\mathbf{p}_k$ of the virtual LRF is tested, and the deviation between the virtual and actual readings is computed as

$$\delta_k = \sqrt{\frac{\sum_{j=1}^{n_r} (s_j - \sigma_j)^2}{n_r}}, \tag{11}$$

where s_j is the average value (over three readings, as mentioned above) along ray j of the actual LRF, and σ_j is the average reading along the same ray for the virtual LRF. n_r denotes the number of (used) rays for the LRF. In principle all $N_r = 682$ rays could have been used. However, if all rays were to be used, the simulation would run rather slowly. Thus, a smaller number of rays (see Subsect. 6.1) was used. Of course, this implies no

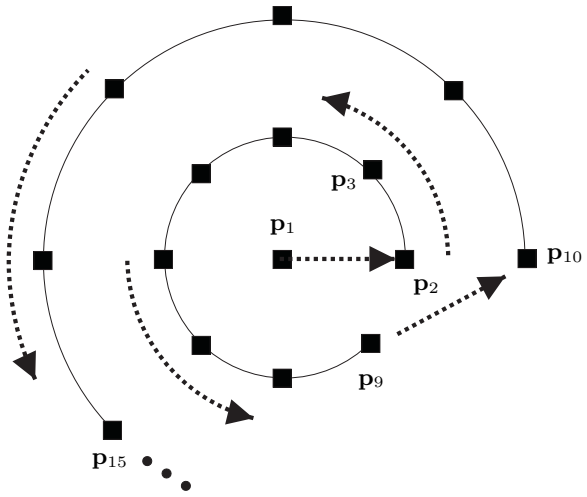


Fig. 6 A sequence of positions $\mathbf{p}_k$ for the placement of the virtual LRF carried out by the *localization* behavior, illustrated for the case $\Delta_\phi = \pi/4$. At each point $\mathbf{p}_k$, the virtual LRF generates an average over three readings (based on the map), and compares them to the reading obtained from the real LRF to form a deviation δ_k. The first point $\mathbf{p}_1$ coincides with the estimated position $\hat{\mathbf{x}}$ at the moment of activation of the *localization* behavior.

[4] Thus, the *localization* behavior assumes implicitly that the odometry has not drifted too much. Note, however, that it is up to the behavioral organizer to activate the *localization* behavior at the correct time.

[5] Note that, even though the compass has limited accuracy it does not, of course, suffer from a systematic drift (increasing errors) like the odometry.

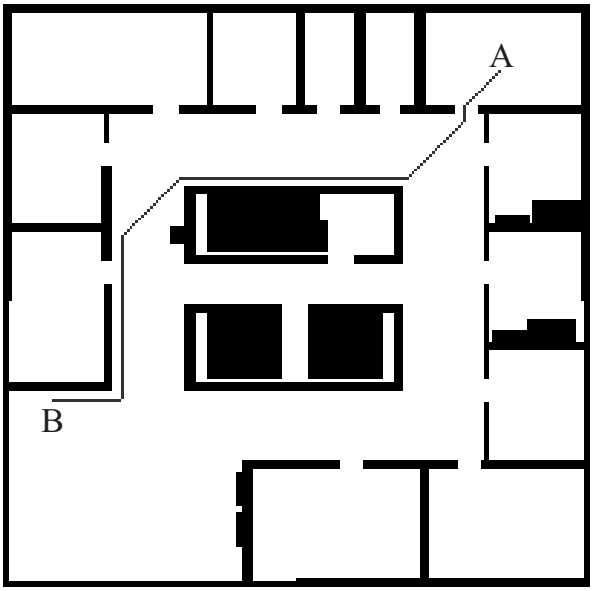


Fig. 7 Path between the initial position (A) and the navigation target used during the optimization of the localization behavior (see Sect. 6.1).

restriction; on the contrary, if the behavior can be made to function well even when only a small fraction of the available rays are used, it would perform even better (or, at least, equally well) if a larger number of rays were to be used in the implementation of the behavior in a real robot.

It should be noted that the localization procedure can be interrupted at any time, if the utility of the *localization* behavior should drop below that of any other behavior. Thus, the procedure just described may not always result even in a modified position estimate, let alone a *better* estimate than what was available before activation of *localization*. If the behavior is active for $T_{\rm loc}$ s, the number of virtual LRF positions (K) tested can be obtained from the equation

$$T_{\rm loc} = T_s + 3T_{\rm LRF} + K{\rm d}t, \tag{12}$$

where dt is the length of the time step in the simulation. Note that the value of T_s depends on the speed of the robot upon activation of *localization* and may thus vary between activations of this behavior. Note also that if the behavior is only active for a time shorter than $T_s+3T_{\rm LRF}$ s, no odometric recalibration will occur at all.

During step 3 of the matching procedure, the robot will thus generate a sequence of deviations δ_k for the various tested positions, starting from δ_1, i.e. the deviation obtained at the current estimated position $\hat{\mathbf{x}}$. If, for some k, a smaller deviation δ_k than δ_1 is found, the robot recalibrates its estimated position, by setting $\hat{\mathbf{x}} = \mathbf{p}_k$. If no virtual position $\mathbf{p}_k$ leads to a smaller deviation than δ_1, the robot does not update its estimated position.

5.3 Obstacle avoidance

In contrast to the two behaviors just described, the *obstacle avoidance* behavior is very simple: When activated by the behavior selection system, this behavior simply sets the speed of the motors to equal, negative values, so that the robot will move backwards in a straight line. Note that it is the job of the behavior *selection* system both to activate the behavior, by raising the corresponding utility value as a result of, for example, a non-zero reading of one or several touch sensors, and to de-activate it by lowering its utility as soon as the touch sensors no longer are in contact with an obstacle.

6 Results

Behavior selection systems were evaluated in simulations carried out using a simulator based on the UFLib simulation library. In each simulation, the robot was allowed to move for $T = 150$ s. The time step length was set to d$t = 0.01$ s, and simulations were terminated if the body of the robot collided with an object (e.g. a wall), but not, of course, in cases where only the touch sensors were in contact with the object. In each time step, the values of the five state variables x_1, x_2, s_1, s_2, and s_3 were obtained, and the utility values U_1, U_2, and U_3 were calculated. The robot then activated (or kept active) the behavior with the highest utility value. Note that the variation of the hormone variables x_1 and x_2, as well as the parameters of the utility functions, were obtained from the two chromosomes illustrated in Fig. 5, and thus varied from individual to individual.

However, before the evolution of a complete behavioral organizer was attempted, it was necessary to investigate the performance of the individual behaviors. B_1 (*path navigation*) and B_3 (*obstacle avoidance*) could be tested quite easily. However, for B_2 (*localization*), a more thorough study, aimed at finding appropriate values for the parameters Δ_r, Δ_ϕ, and n_r, was needed. A description of those tests is given in Subsect. 6.1. Next, in Subsect. 6.2, a description is given of the results obtained from runs aimed at generating a behavior selection system.

6.1 Optimization of the localization behavior

The localization behavior described in Subsect. 5.2 above depends on three parameters, namely Δ_r, Δ_ϕ, and n_r. In order to test the behavior, and to optimize the values of these parameters, simulations were made in which only the two behaviors *path navigation* and *localization* were used, and where the switch between behaviors was hard-coded such that, in each 12 s period, the *path navigation* behavior was executed for the first 10 s, and the *localization* behavior for the final 2 s. Thus, the robotic brain was fixed, and (for given values of Δ_r, Δ_ϕ, and

Δ_r	Δ_ϕ	n_r	$\hat{p}$
0.025	$\pi/2$	34	0.82
0.025	$\pi/4$	34	0.86
0.025	$\pi/2$	68	0.77
0.025	$\pi/4$	68	0.85
0.015	$\pi/2$	34	0.82
0.015	$\pi/4$	34	0.88

Table 2 The table shows the estimated probability $\hat{p}$ of an improvement in the position estimate, as a result of running *localization* for 2 s, after a 10 s navigation period. Quantitative measures of the improvements are given in Table 3.

n_r) the only difference between successive simulations was caused by the noise in the motors and sensors. Each simulation lasted 120 s, i.e. 10 complete cycles of navigation and localization. Thus, in this case, there was no need to use an EA, and evolutionary optimization was therefore temporarily disabled here.

The initial position, and the navigation target, are illustrated in Fig. 7. For each setting of the parameters Δ_r, Δ_ϕ, and n_r, many simulations were carried out, in order to obtain reliable estimates of the average improvement (if any) in the estimated position obtained from the localization behavior. Note that, with $T_s \approx 0.10 - 0.40s$ and $T_{\mathrm{LRF}} = 0.10s$, the robot will, according to Eq. (12) have time to check around $K = 130 - 170$ different positions during each 2 s activation period of *localization*.

The results obtained for various values of Δ_r, Δ_ϕ, and n_r are shown in Tables 2 and 3. The number of rays used (n_r) was set either to 34 (5 % of the total number of available rays) or to 68 (10 % of the total). Table 2 shows the probability $\hat{p}$ of an improvement of the position estimate as a result of running the *localization* behavior. In each case, the value of $\hat{p}$ is based on at least 100 navigation and localization cycles. As is evident from the table, the *localization* behavior does quite well, improving the estimate of the robot's current position in $77-88\%$ of all attempts. It should also be noted that failed attempts, i.e. those that lead to a larger position error than before *localization* was executed, often occured when the initial estimate was quite good, and the absolute magnitude of the position error, both before and after localization, turned out to be quite small in those cases.

Table 3 shows some more detailed results concerning the improvement of the position estimates. In this table, the average position estimate (after localization) is shown, as a function of the position estimate before localization. The numerical entries in the table are based on averages of at least five measurements and, in some cases, up to 20 measurements. However, the typical drift of the odometry (with the noise settings used here, see Table 1) over a 10 s activation of *path navigation* was around 0.10-0.30 m. Thus, for some cases the ~ 100 navigation and localization cycles did not generate a sufficient number of measurements[6] (i.e. at least five) to form a reliable average for certain values of the position error $|\mathbf{x}_e|$.

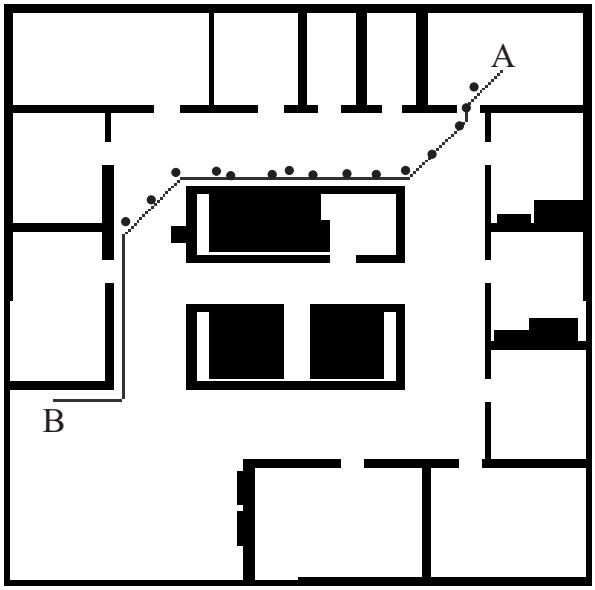


Fig. 8 A typical path followed by a robot in Run 1. The dots indicate the positions at which the robot activated its *localization* behavior, temporarily suspending its navigation along the intended path (solid line).

From the table it can be seen that the *localization* behavior generally improves the position estimate. In addition to the values shown in the table, a few measurements were obtained also for $|\mathbf{x}_e| = 0.50 \pm 0.05$ m. In *all* such cases (16 in all, distributed over the six different parameter settings), the position estimate was improved.

Furthermore, Table 3 shows that the best results were obtained for $\Delta_\phi = \pi/4$, i.e. in cases where 8 (rather than 4) positions are checked on each circle (see Fig. 6), a result which is also evident from Table 2.

6.2 Optimization of the behavior selection system

Once appropriate settings had been obtained for the *localization* behavior, namely $\Delta_r = 0.025$, $\Delta_\phi = \pi/4$, and $n_r = 32$, simulations aimed at generating a behavior selection system was carried out. A large number of runs were completed and, in most cases, the EA was able quickly to optimize the utility functions, and the variation of the hormone variables, so that the robot managed to navigate accurately and rather efficiently in the arena. Each behavior selection system (i.e. each individual in the EA) was evaluated for 10,000 time steps, corresponding to 100 seconds.

The fitness measure was taken simply as the number of waypoints (see Subsect. 5.1) reached by the robot dur-

[6] This does not, of course, indicate a failure of the *localization* behavior; instead, it merely indicates the magnitude of the typical odometric drift during 10 s of navigation.

Δ_r	Δ_ϕ	n_r	$\lvert\mathbf{x}_e\rvert = 0.05 \pm 0.005$	$\lvert\mathbf{x}_e\rvert = 0.10 \pm 0.01$	$\lvert\mathbf{x}_e\rvert = 0.20 \pm 0.02$	$\lvert\mathbf{x}_e\rvert = 0.30 \pm 0.03$
0.025	$\pi/2$	34	N/A	0.078	0.116	0.176
0.025	$\pi/4$	34	0.024	0.071	0.063	0.084
0.025	$\pi/2$	68	0.043	0.079	0.128	0.161
0.025	$\pi/4$	68	0.046	0.069	0.075	N/A
0.015	$\pi/2$	34	0.048	0.077	0.128	0.152
0.015	$\pi/4$	34	N/A	0.055	0.124	0.114

Table 3 Columns 4-7 show the average position error (unit: meters) *after* re-calibration (i.e. after executing *localization*) as a function of the magnitude of the position error $|\mathbf{x}_e| = |\mathbf{x} - \hat{\mathbf{x}}|$ *before* re-calibration. The value of Δ_r is given in meters, and the value of Δ_ϕ in radians.

ing its evaluation. The starting point and the navigation target were the same as in the test of the *localization* behavior above, i.e. the points shown in Fig. 7. Note, however, that the presence of noise, both in sensors and motors, made sure that no two evaluations were exactly identical.

The performance of the best robots obtained from the EA was investigated by means of re-evaluations, in which the robot was placed in a random position in the arena, and was then required to reach a randomly chosen target point. If the target point was reached, a new target point was generated randomly, etc.

A typical path, followed by the best individual in a representative run (hereafter called Run 1), is shown in Fig. 8. The solid line shows the intended path, obtained by the *path navigation behavior*. The dots show some of the actual locations reached by the robot during the first 100 s of its motion, namely those locations at which the robot chose to activate its *localization* behavior. As can be seen from the figure, the robot prioritized navigation accuracy over speed, and carried out a total of 15 localizations during this time interval. The average duration of the localization episodes was 1.8 s. Thus, the robot spent a total of 27 s (27%) on localization.

The variation (with time) of the utility functions during the first 20 s of this simulation are shown in Fig. 9, and the variation of the hormone levels during the same time interval are shown in Fig. 10. The time constants for the hormone variables ranged from around 7 s to around 27 s.

Note that the *obstacle avoidance* behavior was never activated in this simulation. In fact, the combination of *path navigation* and *localization* turned out to work very well, and the *obstacle avoidance* behavior was therefore only needed very rarely. However, the EA still managed to optimize the behavior selection system in such a way that *obstacle avoidance* could be activated on those rare occasions when it actually was needed. An example of the variation in the utility functions from a re-evaluation of the best individual in Run 1 is shown in Fig. 11; as can be seen in the figure, the *obstacle avoidance* behavior (dotted line) was activated for a brief instant around $t = 39.5$ s, when the robot slightly touched the wall.

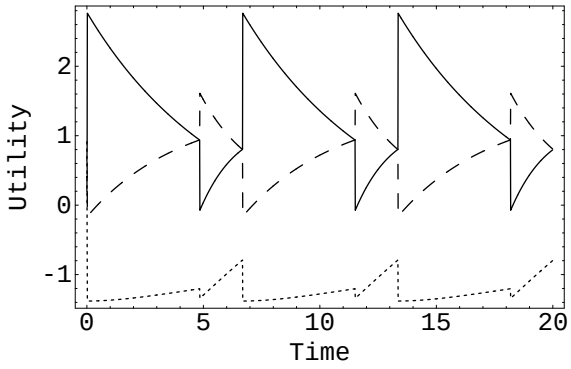


Fig. 9 Variation in the utility values during the first 20 s of a re-evaluation of the best individual found in Run 1. U_1, U_2, and U_3 are shown as solid, dashed, and dotted lines, respectively.

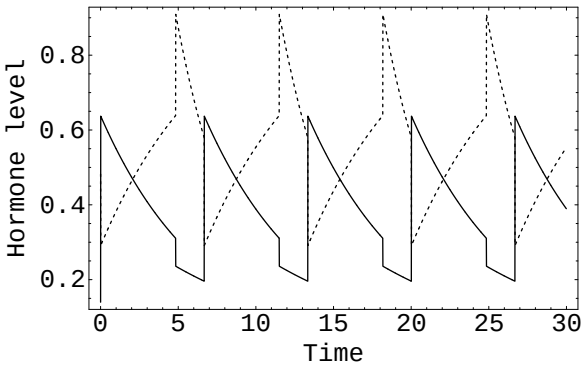


Fig. 10 Variation in the hormone values during the first 20 s of a re-evaluation of the best individual found in Run 1. The solid line shows x_1, and the dotted line shows x_2.

7 Discussion

An interesting, and somewhat surprising, observation is that, despite its simplicity, the *localization* behavior worked quite well. The detailed performance of the behavior was also rather insensitive to the particular parameters chosen (see Table 3). Furthermore, due to the noise in the odometry, the robot could not always determine accurately whether or not it was moving; in fact, in many, if not most, activations of the *localization* behavior, the robot was moving slowly forward *after* the initialization of the scan matching procedure (step 3 in

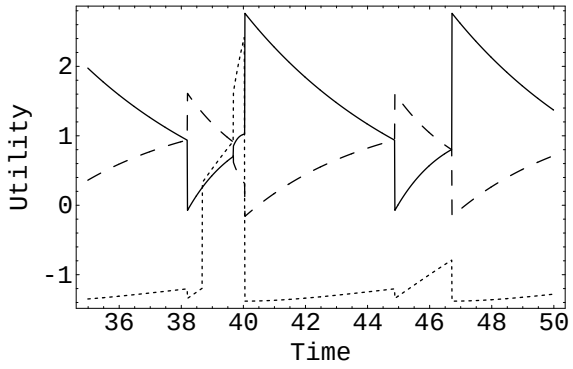


Fig. 11 Variation in the utility values between $t = 35$ and $t = 50$ for a re-evaluation of the best individual found in Run 1. U_1, U_2, and U_3 are shown as solid, dashed, and dotted lines, respectively.

the description of the behavior, see Subsect. 5.2), making the performance of the *localization* behavior even more impressive.

As for the scan matching, one may wonder what processor speed would be needed in order to carry out one complete virtual LRF reading, and the associated scan matching, in one time step (0.01 s). The reading of the virtual LRF requires that, for each of the n_r rays, the intersection between a ray and the nearest object should be determined. The whole process should then be repeated three times (to form an average reading, as described in Subsect. 5.2). Finally, the deviation δ_k should be computed. It turns out that a 2.5 GHz computer can execute around 200 such steps (three readings and a matching, using $n_r = 34$) in one second. Thus, in order to carry out *one* step in 0.01 s, a clock frequency of around 1.25 GHz would be needed, i.e. well within the limits of what is possible, at least if the robotic brain is implemented in a PC (e.g. a laptop) placed on the robot.

For the behavior selection problem, it is interesting to note that the activation of *localization* involves a trade-off: if the behavior is activated very infrequently, the robot suffers the risk of deviating from its path and, possibly, colliding. Also, if the deviation between the actual and the estimated position becomes too large, the *localization* behavior may be unable to correct the error. On the other hand, if *localization* is activated very often, the robot will navigate quite slowly, spending too much time just determining its position. As for the *duration* of the *localization* behavior, it must be set long enough so that the behavior has time to search out to a sufficiently large radius (see Fig. 6).

In the UF method, the frequency of activation, as well as the duration, of the behaviors, including *localization* is determined, of course, by the utility functions and the hormone variables. One of the main advantages with the method is its ability to handle trade-offs of the kind just described. In the particular case considered here, the EA was able to find, quite quickly, appropriate utility functions and hormone variations so as to activate the *localization* behavior at the right moments, and to keep the behavior active for just the right amount of time. Similarly, the behavior selection system was able to activate (albeit very briefly) the *obstacle avoidance* behavior, when needed.

As for the number of hormones, the choice of $M = 2$ was quite arbitrary. The main idea was to allow the behavior selection system to use one hormone for representing fear (i.e. in order to activate *obstacle avoidance*), and one for representing confusion (i.e. in order to activate *localization*). However, the EA was in no way *required* to use the hormone variables in this way. It should also be noted that any number of hormones could have been added; the EA can always choose to ignore hormones that are not needed, simply by setting x^{in}, x^{out}, and Γ to 0.

8 Conclusion and further work

In this paper, it has been shown that the (modified) UF method, incorporating evolutionary optimization of both utility functions and hormone variables, is able to generate a successful behavior selection system for a real-world task (albeit somewhat simplified, in this study). The evolved behavior selection systems are able to switch between a *path navigation* behavior and a *localization* behavior at appropriate times in order to achieve an overall goal of navigating between two arbitrary points in a given arena. Furthermore, it has been shown that a rather simple *localization* behavior, based on laser range finder scan matching works quite well, despite its simplicity.

In future work, the *localization* behavior will be optimized further, possibly by using an EA to search through the space of parameters. Furthermore, the simple dynamics of the hormone variables may be extended to involve feedback, rather than the simple temporal variation that has been used in this paper. The duration of the active periods for the *obstacle avoidance* behavior can perhaps be extended if the touch sensor readings are given an artificial, non-zero decay time, rather than the instantaneous drop from 1 to 0 used here. Such a decay time would be easy to implement both in simulation and in a real robot. Furthermore, moving objects (e.g. people) will be added in the simulations, to increase the degree of realism.

Finally, and most importantly, the results obtained here will be transferred to a real robot. A simple, scale 1:3 prototype has already been built and the construction of a full-scale prototype will begin in the spring of 2007.

Acknowledgments

The study presented in this paper is part of a cooperative research project, aimed at constructing a transportation robot. The project involves researchers from Waseda University (Tokyo, Japan), Tsukuba University (Tsukuba, Japan), Future University (Hakodate, Japan), and Chalmers University of Technology (Göteborg, Sweden). The authors wish to thank Prof. Shuji Hashimoto, Drs. Ryo Saegusa, Pitoyo Hartono, and Kenji Suzuki, and Mr. Krister Wolff, for helpful discussions during the preparation of this paper. Finally, the authors wish to thank the Carl Trygger foundation for financial support.

References

1. R. C. Arkin, *Behavior-based robotics*. Cambridge, MA: The MIT Press, 1998.
2. M. Wahde, *An introduction to adaptive algorithms and intelligent machines*, 5th ed. Göteborg: Chalmers Reproservice, 2006.
3. R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, 1986.
4. P. Maes, "How to do the right thing," AI-Laboratory, Massachusetts Institute of Technoglogy, Cambridge, MA, Technical Report NE 43-836, 1989.
5. B. Blumberg, "Action-selection in Hamsterdam: Lessons from ethology," in *Proceedings of the Third International Conference on the Simulation of Adaptive Behavior*, Brighton, England, August 1994, pp. 108–117.
6. J. Rosenblatt, "Damn: A distributed architecture for mobile navigation," in *AAAI 1995 Spring Symposium on lessons learned for implemented software architectures for physical agents*, March 1995, pp. 167–178.
7. P. Pirjanian, "Behavior-coordination mechanisms — State-of-the-art," Institute for Robotics and Intelligent Systems, University of Southern California, Technical report IRIS-99-375, October 1999.
8. D. McFarland and E. Spier, "Basic cycles, utility, and opportunism in self-sufficient robots," *Robotics and Autonomous Systems*, vol. 20, pp. 179–190, 1997.
9. M. Wahde, "A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions," *Journal of Systems and Control Engineering*, vol. 217, no. 4, pp. 249–258, September 2003.
10. J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, 3rd ed. Princeton, N. J.: Princeton University Press, 1953.
11. H. Chernoff and L. Moses, *Elementary decision theory*. New York, NY, USA: Dover Publications, Inc., 1986.
12. D. McFarland and T. Bösser, *Intelligent behavior in animals and robots*. The MIT Press, 1993.
13. M. Wahde, J. Pettersson, H. Sandholt, and K. Wolff, "Behavioral selection using the utility function method: A case study involving a simple guard robot," in *Proc. of the* 3rd *Int. Symp. on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, 2005, pp. 261–266.
14. J. Pettersson, D. Sandberg, K. Wolff, and M. Wahde, "Behavioral selection in domestic assistance robots: A comparison of different methods for optimization of utility functions," in *Proceedings of the 2006 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2006)*, Taipei, Taiwan, October 2006, (To appear).
15. J. Pettersson and M. Wahde, "Improving generalization in a behavioral selection problem using multiple simulations," in *Proceedings of the Joint* 3rd *International Conference on Soft Computing and Intelligent Systems and* 7th *International Symposium on advanced Intelligent Systems (SCIS & ISIS 2006)*, Tokyo, Japan, September 2006, pp. 989–994, (To appear).
16. C. Kee *et al.*, "Development of indoor navigation system using asynchronous pseudolites," in *Proceedings of ION GPS-2000*, 2000, pp. 1038–1045.
17. M. Wahde and J. Pettersson, "A general-purpose transportation robot: An outline of work in progress," in *Proceedings of the* 15th *IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 06)*, Hatfield, United Kingdom, 2006, pp. 722–726, (To appear).
18. S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. The MIT Press, September 2005.
19. U. Frese, "A discussion of simultaneous localization and mapping," *Autonomous Robots*, vol. 20, no. 1, pp. 25–42, 2006.
20. C. F. Olson, "Probabilistic self-localization for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 16, no. 1, pp. 55–66, February 2000.
21. M. O. Franz, B. Schöllkopf, H. A. Mallot, and H. H. Bülthoff, "Where did i take that snapshot? scene-based homing by image matching," *Biological Cybernetics*, vol. 79, pp. 191–202, 1998.
22. M. Wahde and J. Pettersson, "UFLibrary tutorial," available at: http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary/.
23. ——, "UFLibrary demo," available at: http://www.me.chalmers.se/~mwahde/robotics/UFMethod/UFLibrary/Demo.html%.
24. J. Pettersson and M. Wahde, "Uflibrary: A simulation library implementing the utility function method for behavioral organization in autonomous robots," *Int. J. on Artificial Intelligence Tools*, 2005, (Submitted).
25. ——, "Application of the utility function method for behavioral organization in a locomotion task," *IEEE Trans. Evol. Comp.*, vol. 9, no. 5, pp. 506–521, 2005.
26. P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths in graphs," *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, July 1968.
27. S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2002.