NERD
Neurodynamics and Evolutionary Robotics Development Kit

Christian Rempis, Verena Thomas, Ferry Bachmann, and Frank Pasemann

Institute of Cognitive Science
University of Osnabrück, Germany
{christian.rempis,frank.pasemann}@uni-osnabrueck.de,
thomas@cs.uni-bonn.de,bachmann@informatik.hu-berlin.de
http://ikw.uni-osnabrueck.de/~neurokybernetik/

Abstract. The aim of Evolutionary Robotics is to develop neural systems for behavior control of autonomous robots. For non-trivial behaviors or non-trivial machines the implementation effort for suitably specialized simulators and evolution environments is often very high. The *Neurodynamics and Evolutionary Robotics Development kit* (NERD), presented in this article, is a free open-source framework to rapidly implement such applications. It includes separate libraries (1) for the simulation of arbitrary robots in dynamic environments, allowing the exchange of underlying physics engines, (2) the simulation, manipulation and analysis of recurrent neural networks for behavior control, and (3) an extensible evolution framework with a number of neuro-evolution algorithms. NERD comes with a set of applications that can be used directly for many evolutionary robotics experiments. Simulation scenarios and specific extensions can be defined via XML, scripts and custom plug-ins. The NERD kit is available at nerd.x-bot.org under the GPL license.

Key words: Simulation, Evolutionary Robotics, Neuro-evolution

1 Introduction

The development of behavior controllers for autonomous robots with methods from the field of evolutionary robotics is by now well established [4][5][7]. Instead of analytical construction, evolutionary techniques – sometimes together with other learning methods – are used to search for suitable solutions. Inspired by the properties of biological brains, artificial recurrent neural networks turned out to be effective control structures, especially for more complex behaviors. With respect to evolutionary algorithms they are also suitable because of their simple (syntactical) structure that allows the construction and adaptation of controllers with a set of simple modification operators.

Usually controllers for physical robots are evolved in simulation, (1) because evolutionary approaches and learning methods require experiments with many comparable iterations, (2) to protect the physical robot from wear-out or evolved, potentially harmful control, and (3) to speed up evolution by running the simulation faster than real-time, for instance on a computer cluster. Furthermore (4) simulations are useful to reduce cost, e.g. by evolving multi-agent systems with way more robots than are actually available in a lab, or by co-evolving morphological aspects that may lead to structural enhancements to the physical robots without the need to build all intermediate steps in hardware.

However, using a simulator for evolution may also lead to problems: Evolution usually optimizes its solutions exploiting all the details of a simulated robot, including modeling differences and simplifications. Such controllers are often difficult or impossible to transfer to the targeted physical robot, because the differences are too large. Therefore critical elements of the robot simulation have to be modeled accurately, often very specifically for a given robot or experiment. But this is sometimes difficult to accomplish with closed-source applications or with simulators having only a fixed set of components. Furthermore, evolution of controllers is usually an iterative process in which the scenarios, environments and the robots are iteratively modified and changed until a suitable experiment is defined. Thus, a simulator should be simple to adapt and it should be extensible also by very specific models. Hereby it should simplify the reuse of frequently used program parts to speed up the simulator development.

The evolution or optimization techniques used to evolve neuro-controllers can – and should – not be fixed because the development of non-trivial control networks often requires the utilization of new or adapted algorithms. Keeping the actual evolution algorithm open to be chosen freely by the user, while allowing the reuse of an entire integration framework – such as logging, statistics calculations, configuration, user interfaces, computer cluster support, evaluation and analysis tools – facilitates the rapid realization of new algorithms.

The *Neurodynamics and Evolutionary Robotics Development kit* (NERD) [15], described in this publication, is a framework to rapidly create simulations and applications in the context of evolutionary robotics. Its design focuses on extensibility and flexibility to allow for a wide variety of control applications without the need to re-implement commonly used features. NERD is open source (under GPL license) and comes with a number of ready-to-use applications for simulators, neural network analysis and evolutionary algorithms. Providing libraries instead of only fixed applications allows users to build upon existing code to create their own, specialized neural control applications, without being forced to use a specific physics engine, rendering engine, controller type, genome representation or evolutionary algorithm. However, the provided applications themselves are already flexible enough to allow the definition of complex evolutionary experiments without or with only little programming skills.

The library features a flexible application core with plug-in support, object management and network interfaces, a simulation library with exchangeable physics engine, implementations of neuro-evolution algorithms, an extensible neural network model, an interactive editor for neural networks, computer cluster support, and optional graphical user interfaces for visualization and configuration. Details are described in the remaining chapters.

The NERD kit is under continuous development since 2007 and is used for experiments in scientific and educational areas, such as motion control of humanoid robots, biologically inspired behaviors of animats and in teaching evolutionary robotics and neuro-dynamics.

2 Concepts and Library Organization

2.1 Library Organization

The NERD library was primarily developed for the structure evolution of neural networks serving as controllers for physical robots. Nevertheless the library was designed to be as universal and extensible as possible. This is important, because at the beginning of a project it is often difficult, if not impossible, to know all eventual requirements in advance. In the field of evolutionary robotics, this is especially true for the accurate simulation of rigid body dynamics, the optimization algorithms and the neural network models.

Table 1. NERD Libraries and Dependencies

Library	**Description**	**Dependencies**
core	Basic core library, GVR, events, plug-ins, configuration, logging, basic GUI	none
simulator	Simulation components, simulation visualization and physics abstraction layer	core
odePhysics	ODE implementation of the physics abstraction layer	core simulator libOde (external)
ibdsPhysics	IBDS implementation of the physics abstraction layer	core simulator libIbds (external)
neuralNetwork	Neural network models and components	core
networkEditor	Graphical network editor and analysis tools	core neuralNetwork
evolution	Evolutionary algorithm framework	core
evaluation	Components to evaluate individuals of the evolution	core evolution
neuroEvolution	Neuro-evolution algorithms	core evolution neuralNetwork
neuroSimEvaluation	Evaluation library for neural networks on simulated agents	core simulator evaluation

The libraries are written in standard C++ and make use of the QT library (Version 4.5) [9]. QT is a C++ framework, that hides platform dependent functionality – like networking, file management, graphical user interfaces and OpenGL rendering – behind a platform independent API. This allows NERD applications to be compiled for Windows, Linux and Mac without modifications. Apart from optional physics libraries for the simulation of rigid body dynamics no additional external libraries are required.

The framework can be extended by additional, custom libraries or by plug-ins to be loaded at runtime to extend applications with custom elements. The NERD kit also comes with a number of concrete applications (Sect.3), that bundle the functionality of the libraries for specific tasks. Users of the NERD kit may either implement custom applications, or use the enclosed applications and extend their functionality via plug-ins, XML and scripts to fit the application to a specific scenario.

2.2 Basic Features

System Core. All NERD applications are built on top of a single, global core object, accessible as a singleton from everywhere in the code. All objects, managers and extensions should be registered during their creation at the core to be initialized, set up, shut down and destroyed properly. Objects can be registered as *global* objects by giving them unique names. Such objects are accessible via the core from any other part of the code using their name. This makes it easy to add and use shared objects.

Global Variable Repository. The global variable repository (GVR) implements a blackboard architecture that allows components of an application to exchange information without actually knowing each other directly. This makes it easy to add or exchange software components, even if they are very different from each other, because the actual class or interface type of each component is not of relevance (Fig. 1). What matters is only the variable interface provided to the GVR. All software components can register local variables at the GVR, hereby giving a global name to the local variable. The names are organized like directory names, dividing the global name-space into a hierarchy of smaller name-spaces. During startup of the application, all components may *bind* to variables of the GVR by name, hereby obtaining a pointer to each desired variable object. With this pointer a component can read or write to and from this variable during the rest of the application lifetime without performance loss. The GVR is registered as global object at the core.

Variables in the GVR also implement a notification mechanism. With this, objects register as observer to selected variables and can react on changes of variables immediately.

Each variable object also provides methods to read and write the variable using strings. Therefore loading, saving, displaying, transmitting and manipulating variables is possible in a type-independent unified way. New components do not need to implement specific mechanisms for variable manipulation and configuration, because this is already provided by the NERD standard tools.

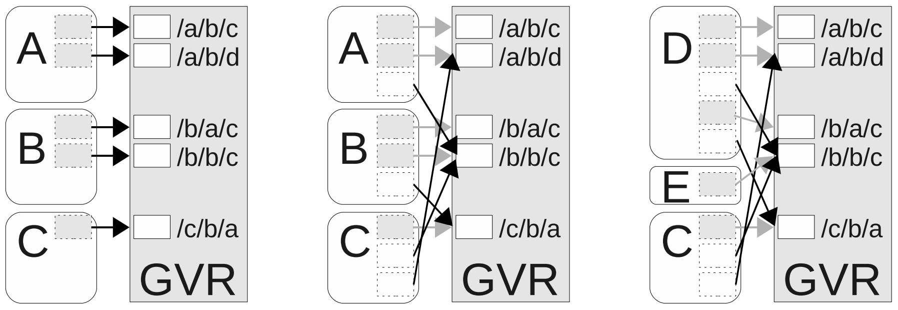


Fig. 1. Global Variable Repository. Left: Three components registering local variables at the GVR. Mid: Components bind to variables of other components using the global names of the variables (black arrows). Right: Components A and B are replaced by D and E. C works without changes, even if the replaced components are very different.

Events. NERD applications are event driven. These events are provided by the so called event manager, that is registered as global object at the core. Any component can use the event manager to create global events or to register for events of other components. Events can be triggered by any component, leading to a notification of all components registered for this event. Thus, all components are loosely coupled through events and contribute their functionality as reaction to these. Adding a new component is simple: The component just has to register for a specific existing event to do the right thing at the right time without even knowing the entire context of the application. NERD applications typically consist of a number of such separate components – connected by the GVR and global events – and an application dependent event loop, that triggers the events in the correct order.

Standard Graphical User Interface. The core library provides a set of optional GUI elements to control, visualize, customize and debug applications. These include a property editor to load, save, modify and observe variables in the GVR, plotters to observe variables over time and loggers.

2.3 Simulation

The optional simulation library was designed to provide all common tools and mechanisms to simulate autonomous mobile robots in dynamic environments.

Physics Abstraction Layer. Special emphasis has been placed on the accuracy of the robot simulation. The rigid body dynamics of a simulation usually is processed by external physics libraries [11][1][2], that all have their strengths and weaknesses concerning accuracy and performance. Because our goal is the evolution of neuro-controllers for physical robots, the simulation of the actual behaviors has to be as accurate as possible, so that evolved controllers can be transferred to the physical machine. Therefore NERD does not depend on a single, fixed physics engine. Instead all simulated objects are represented within a physics independent abstraction layer. The concrete implementation of the physics simulation and collision handling is detached from the rest of the simulation system, so that a replacement of the physics engine is possible with little impact on the other parts of the system. NERD therefore remains open for enhancements of existing engines, new approaches or commercial systems when this is required for the simulation of a particular scenario.

Currently the abstraction layer provides prototypes for boxes, spheres, capsules, rays, different kinds of joints and motors, light sources, and sensors for light, acceleration, forces, contacts, joint angles and distances.

Environment XML. Simulation scenarios are described in XML files that are loaded at runtime. The XML describes and configures all objects in the simulation. Different kinds of randomization techniques are supported to randomize all object parameters in a deterministic way, i.e. that all individuals of a generation during evolution face the same randomized parameter sets. The XML can also be used to set parameters in the GVR, to allow or prevent collisions between objects, and to add so called *collision events*. The latter are events that are triggered when a certain set of objects collide, which is a useful information for fitness functions.

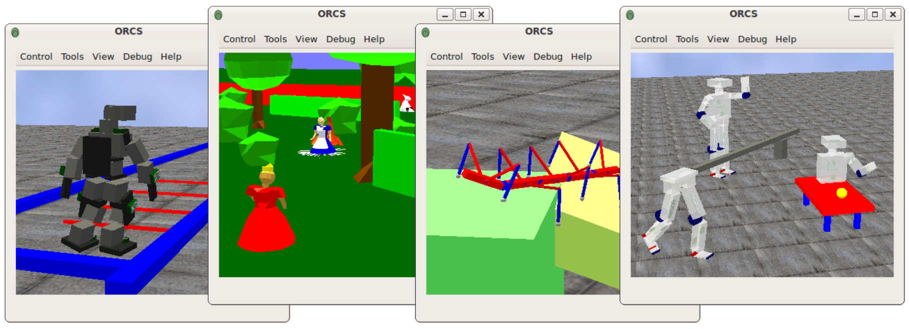

ORCS
Control Tools View Debug Help
ORCS
Control Tools View Debug Help
ORCS
Control Tools View Debug Help
ORCS
Control Tools View Debug Help

Fig. 2. Example Environments: (1) A-Series humanoid walking scenario (2) A-life experiment "Alice in Wonderland" (3) A walking machine with variable number of segments (4) Experiments with the modular M-Series humanoid

Agent Control. Agents are defined by grouping simulated objects together. Each agent automatically collects all motors and sensors of its body and provides an interface for behavior controllers. These interfaces provide a simple collection of typed, named variables, that can be accessed by plug-ins to read out the sensors and to control the actuators. Furthermore these variables are available through the GVR and thus allow manipulations in many different ways. With this, in principle, any kind of control is possible by adding simple plug-ins with control code, mapping the input of the interface to its output with any desired approach. However, the main control paradigm in NERD is the control with recurrent neural networks, which is supported by an extensible framework (Sect. 2.5). There the neurons of the input and output layer of a neural network are automatically connected to the interface variables, mapping between native variable values and neural activations.

A useful standard tool in NERD are the so called *control sliders*, which allow the control of any actuator of an agent using sliders. This helps to test robot models or to get a feeling for sensor and motor ranges and the overall body behavior of a robot.

The NERD simulation can also be used as pure simulation environment, running the control and perception of a robot in separate applications. This is beneficial if one wants to study the behavior of a physical robot with respect to an already given control application. In this case NERD and its agents may be controlled via UDP or other network protocols. NERD comes with a powerful own protocol (Sect. 2.4), that allows to read, stream and write all variables of the GVR and to react to and trigger events. The addition of other communication protocols via plug-ins helps to fit the applications to very specific interface demands.

Visualizations and Camera Support. NERD simulations can be visualized with OpenGL (Fig. 2). NERD supports any number of cameras, either stationary or attached to objects of the simulation, for instance to follow an agent. Each camera allows a separate configuration, e.g. to set the viewing angle, zoom factor, or position. Cameras may be used to observe a simulation scenario from different perspectives, but can also be applied as camera sensors of a robot. Cameras may also output their images to files, so that external vision applications can process the images like normal camera images.

To make the visualizations more lifelike, objects in NERD can have textures and special visualization masks. These masks can have complex shapes, but are considered only for visualization purposes. Thus, the robot will be simulated with simple body shapes for high performance, while the cameras still show the complex shapes of the robot, which is often important for a reliable features detection during vision processing.

2.4 Interfaces and Protocols

Plug-Ins. Existing NERD applications are designed to be extended by plug-ins written in C++. A plug-in just has to be added to a special folder to be loaded at startup, hereby simply attaching its objects to the core, the GVR and the event manager. This allows the extension of an application with arbitrary code. The configuration of plug-ins is simplified with a framework for command line options and the GVR. Thus plug-ins do not need to implement custom configuration interfaces and can simply use the provided standard tools. Plug-ins are frequently used to add custom (motor and sensor) models to the simulation, to extend the neural network model, to add learning rules or evolution algorithms, or to extend the application by communication protocols to interface with external programs.

NERD Standard Protocol (NSP). The NSP is a UDP based communication protocol that allows external applications to access the main coordination parts of NERD, namely the GVR and events. The UDP protocol allows a fast communication on the same computer, where package losses are not to be expected. For control over a network the protocol should be wrapped with the slower TCP protocol to prevent package losses. With the NSP protocol clients can read and write any variable of the GVR and therefore control all main functions of the NERD system. Clients can register to get informed about any change of a variable or to stream its content continuously. Clients can trigger events to drive a NERD application, or receive notifications about triggered events. With this, interacting applications may synchronize with each other and exchange information. Details of the protocol are provided at the website [15].

Specialized Protocols. The integration of existing applications is achieved by implementing their communication protocol. Because of the transparency of the GVR these protocols can access all relevant parts of the NERD environment and simply serve as a mediator between NERD and the communication protocol. With such interfaces, the evolution application may use external simulators (such as the YARS simulator [14]), or a NERD simulator may receive commands from external control programs.

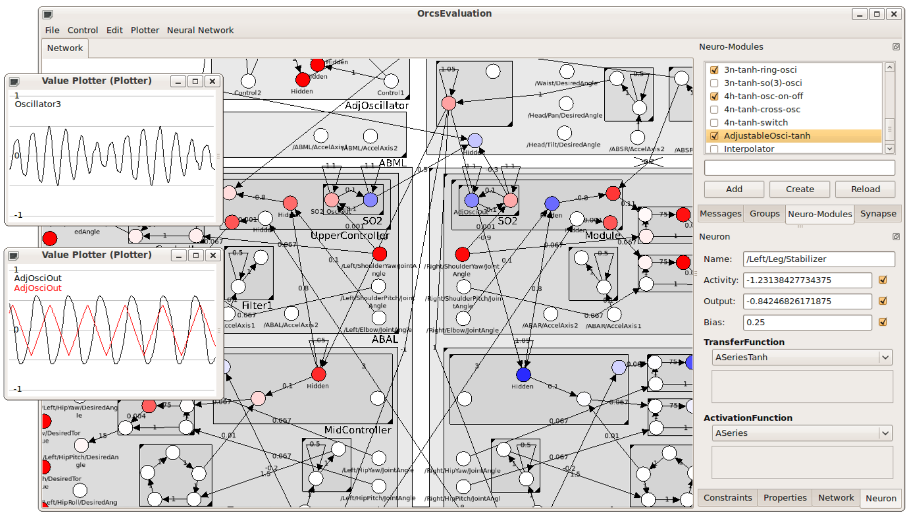
2.5 Neural Network Library

The neural network library provides a flexible, object oriented framework for recurrent neural networks. If the standard model is not sufficient, it can be replaced by custom neuron models via plug-ins. The graphical user interfaces, plotters and manipulation tools even work for most of such extensions. This supports rapid prototyping of network models and the use of the model of choice.

Recurrent Neural Network Model. The NERD network model comprises neurons, synapses, neuron-groups and neuro-modules. The activation and transfer function of each neuron can be chosen and configured separately. The same is true for the model of the synapses. This enables local learning rules at the synapse level or higher-order synapses, i.e. synapses modulating other synapses. Extensions of the set of activation, transfer and synapse functions to adapt the neuron model to specific needs are possible via plug-ins. Each synapse supports the optional control of their execution order, which is useful in feed-forward structures to reduce the delays of neural subnetworks.

Neuron-groups and neuro-modules are part of the *constrained modularization approach* [10] and are used to group and constrain parts of neural networks, as is required by the evolution algorithm ICONE [10]. This allows the automatic enforcement of, for instance, symmetries, topological relations or structure repetitions.

Neural Network Editor. The design of neuro-controllers is supported by a graphical network editor. This editor allows the construction and examination of recurrent neural networks with mouse and keyboard. The editor assists the user with layout and visualization aids to keep even large networks analyzable. Neurons and modules may be named to indicate their function or role. Visual feedback about bias values, interface neurons and other properties, as well as a search function for neurons and synapses by name or weight, help to cope even with large and complex networks.



☑ 3n-tanh-ring-osci
☐ 3n-tanh-so(3)-osci
☑ 4h-tanh-osc-on-off
☐ 4n-tanh-cross-osc
☐ 4n-tanh-switch
☑ AdjustableOsci-tanh
☐ Interpolator

Name: /Left/Leg/Stabilizer
Activity: -1.23138427734375 ☑
Output: -0.84246826171875 ☑
Bias: 0.25 ☑
TransferFunction: ASeriesTanh
ActivationFunction: ASeries

Fig. 3. Neural Network Editor

To ease the construction of neural networks the editor also provides a library of neural building blocks, used to extend a network by functional units instead of only single neurons. Known neural structures, such as oscillators, memory units, filters or controllers, can be reused with this library even without a deep understanding of neuro-dynamics.

The network editor is also a valuable tool to analyze and understand the usually complex evolved neuro-controllers: The activation of each neuron is visualized on-line to support an intuitive comprehension of the network dynamics. Furthermore, pruning experiments – the selective disabling of synapses – in running systems are possible, allowing the experimenter to find the relevant synapses for a behavior in a recoverable way. Also, different plotter types allow the visualization of the activation of any neuron to investigate the role and relation of neurons in the network and the dynamical properties of certain network areas.

Networks in the editor can be exported in numerous output formats, such as scalable vector graphics or native code for physical machines. Additional export formats, e.g. as execution code for a specific robot, may be added via plug-ins.

2.6 Evolution Library

Evolution Framework. The evolution library provides a framework to create custom evolution scenarios. The genomes are not restricted to a specific neural network representation. Thus, the genome representation can be adapted to a certain evolution algorithm or problem, as for instance the co-evolution of body morphology and controller network. Custom evolution algorithms and evaluation strategies can be embedded in the system with little effort, enabling rapid testing of diverse approaches. The framework supports the parallel usage of different evolution algorithms, as well as co-evolution and island models. The evaluation strategy may be freely implemented, e.g. by evaluating each individual separately, by cloning a controller to multiple agents (as in swarms) or by evaluating individuals of a generation or different populations together in the same evaluation scenario (as in predator-prey experiments or with heterogeneous swarms). Also the set of selection methods is extensible via plug-ins and works in combination with most evolution methods. During evolution (customizable) statistics are calculated on the generation and individual level. Statistics information is visualized with the user interface and is logged to files during evolution to allow a detailed analysis of the evolution course.

Fitness Functions. Fitness functions have to be adapted very often until a good strategy is found. Therefore fitness functions can not only be added via C++ plug-ins, but can also be scripted directly in the application (ECMAScript [3]). In both cases the user has full access to all variables of the GVR and events, and herewith to all properties of the simulated environment. These variables can also be modified from within the fitness functions, which allows complex interactions between fitness functions and evaluation scenarios.

3 NERD Standard Applications

The NERD kit comes with a number of pre-build applications for a variety of evolution experiments.

NerdSim is a simulator controlled with the NERD Standard Protocol (NSP) (Sect. 2.4). Thus, its execution is controlled by an external application. The simulation scenario is described with XML. Agents in the simulation are controlled either by an external controller via NSP, or with custom control plug-ins.

NerdNeuroSim is the standard stand-alone simulator for neuro-evolution. Agents are controlled with neuro-controllers or with custom controller plug-ins. This simulator uses the evaluation library to rate the performance of behavior controllers with fitness functions. The simulator also integrates the neural network editor to allow the construction and analysis of neuro-controllers running on the simulated agents.

NerdNeuroEvo is a neuro-evolution application combining the functionality of NerdNeuroSim with the evolution library. The application provides several neuro-evolution algorithms, including ENS3 [8][6], NEAT [12] and ICONE [10]. It features a built-in editor for fitness functions, logging, plotting and real-time observation of evaluations. The NerdNeuroEvo application is especially suited for *interactive neuro-evolution*: The user permanently readjusts the evolution process depending on the current state of the evolution and counteracts undesired local optima by changing the fitness function, the scenario or other aspects of the neural network evolution.

NerdNeuroEvo can also be used with external simulators via network protocols. In this case only the evolution and execution of the neural networks takes place in NerdNeuroEvo, while the physics simulation of the agents is processed in the external simulator.

NerdClusterNeuroEvo is a neuro-evolution application that distributes the evaluation of individuals to a computer cluster using the Sun Grid Engine [13]. The controller evaluation is done in separate instances of NerdNeuroSim, thus NerdClusterNeuroEvo does not include an own simulator. Using a computer cluster speeds up evolution significantly and facilitates interactive evolutions, because the results of parameter or experiment adoptions can be observed much faster.

4 Conclusions

The *Neurodynamics and Evolutionary Robotics Development kit* (NERD) is a free collection of open source libraries and applications for evolutionary robotics experiments. The library provides (1) a core library designed for rapid prototyping and simple integration of new components, (2) a simulator library to simulate arbitrary robots with different physics engines in dynamic environments, (3) a neural network library with a customizable neural network model and a graphical neural network editor, and (4) a general evolution framework with libraries for evolution and evaluation of neural networks with different neuro-evolution algorithms. NERD supports run-time extensions via plug-ins, which allows users to extend existing NERD applications with custom components, such as specific motor and sensor models. In combination with the physics abstraction layer, the simulation accuracy of robots in NERD can be fine controlled to facilitate more similar behaviors between simulated and physical robots. This will ease the transfer of evolved controllers. NERD also supports computer clusters to allow evolutions for complex robots that require simulations with higher accuracy.

Currently the NERD kit is used for the evolution of neural behavior control for different kinds of complex robots, e.g. humanoid robots and multi-legged walking machines.

Acknowledgments. This work was partly funded by EU-Project Number ICT 214856 (ALEAR Artificial Language Evolution on Autonomous Robots. http://www.alear.eu). Thanks go to Chris Reinke, Lucas Theis, Arndt von Twickel and all other contributors of the NERD kit.

References

1. Bender, J. and Schmitt, A. (2006). Fast dynamic simulation of multi-body systems using impulses. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)*, pages 81–90, Madrid (Spain).
2. Bullet Physics Engine. http://bulletphysics.org (last visited May 2010)
3. ECMAScript Language Specification, ECMA-262 Standard, 5th Edition, ECMA International, Rue de Rhone 114, CH-1204 Geneva, December 2009.
4. Floreano, D., Husbands, P., and Nolfi, S. (2008). Evolutionary Robotics. In Siciliano, B. and Khatib, O., editors, *Springer Handbook of Robotics*, pages 1423–1451. Springer.
5. Harvey, I., Paolo, E., Wood, R., Quinn, M., and Tuci, E. (2005). Evolutionary robotics: A new scientific tool for studying cognition. *Artificial Life*, 11(1-2):79–98.
6. Hülse, M., Wischmann, S., and Pasemann, F. (2004). Structure and function of evolved neuro-controllers for autonomous robots. *Connection Science*, 16(4):249–266.
7. Nolfi, S. and Floreano, D. (2004). *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines.* Bradford Book.
8. Pasemann, F., Steinmetz, U., and Dieckman, U. (1999). Evolving structure and function of neurocontrollers. In Angeline, P. J., et al., editors, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 1973–1978, Mayflower Hotel, Washington D.C., USA. IEEE Press.
9. QT Application Framework 4.5. http://qt.nokia.com (last visited May 2010)
10. Rempis C., Pasemann F. (2010), Search space restriction of neuro-evolution through constrained modularization of neural networks. In Madani, K., editor, *Proceedings of the 6th International Workshop on Artificial Neural Networks and Intelligent Information Processing (ANNIIP), In conjunction with ICINCO 2010*, pp. 13–22, Madeira, Portugal, SciTePress.
11. Smith, R.: ODE - Open Dynamics Engine (2007), http://www.ode.org
12. Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural network through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.
13. Sun Grid Engine. http://gridengine.sunsource.net (last visited May 2010)
14. Zahedi, K., von Twickel, A., and Pasemann, F. (2008). YARS: A physical 3D simulator for evolving controllers for real robots. In Carpin et al.: *Simulation, Modeling, and Programming for Autonomous Robots*, Springer, LNAI vol. 5325, 1:75–86.
15. Neurodynamics and Evolutionary Robotics Development Kit. http://nerd.xbot.org (last visited Aug 2010)