Design of an
Artificial Neural Network Simulator
and its Integration with a
Robot Simulation Environment

Diplomarbeit

zur Erlangung des akademischen Grades
"Diplom-Ingenieur"
in der Studienrichtung
Informatik

eingereicht von

Johannes Sixt

Angefertigt am
Institut für Informatik
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

Eingereicht bei
o.Univ.-Prof. Dr. Bruno Buchberger

März 1994

Contents

Foreword **3**

1 Introduction **5**
1.1 Motivation 5
1.2 Neural Networks 6
1.2.1 Theoretical Aspects of Neural Networks 6
1.2.2 What is an ANN? 6
1.2.3 Remarks on Selected Simulators 9
1.3 Robotics 9
1.3.1 Theoretical Aspects 9
1.3.2 Some Remarks about Robot Simulation Systems 14
1.4 Discussion of Some ANN Approaches 15
1.4.1 Some Problems in Robotics Solved with ANN 15
1.4.2 Some Remarks on other ANN Application Fields 16

2 ANN Simulator NETUSE **17**
2.1 The Concept 17
2.1.1 NDL—Network Description Language 18
2.1.2 NTS—Network Test System 20
2.2 NTS—State of the Art 21
2.3 The Command Language of the Monitor Client 22
2.4 The Command Interpreter 23
2.4.1 Commands 23
2.4.2 Parameters 25
2.4.3 Discussion of this Approach 26

3 Robot/NC Simulator SAVE **27**

4 A Concept for a New Version of NETUSE **31**
4.1 Analysis of Experiences 31
4.1.1 Advantages of Earlier Versions 32
4.1.2 Disadvantages of Earlier Versions 33
4.2 The Command Language 33
4.2.1 Syntax and Semantics 33
4.3 The Programming Interface 43
4.3.1 The Structure of an NCI6t Application 44
4.3.2 Naming Conventions 45
4.3.3 An Example of a Class Implementation 45

4.3.4 An Example of a Main Program 50
4.3.5 Initializing NCI6t 51
4.3.6 The Symbol Table 51
4.3.7 Methods 58
4.3.8 Miscellaneous Functions 61

5 Generating and Operating Artificial Neural Networks 63
5.1 User Interface of ANN Classes 63
5.1.1 Generating an ANN 63
5.1.2 Working with an ANN 70
5.2 Implementation of ANN Classes 75
5.2.1 Representation of Neurons and Connections 76
5.2.2 Network Generating Classes 78
5.2.3 Methods for Scheduling 83
5.2.4 Handling Data 86
5.2.5 Methods for Saving and Loading Network States 89

6 The Integrated Simulation Tool at Work: A Prototype 92

7 Conclusions 98

Literature 99

Foreword

This diploma thesis grew out of topics of work in the area of the RISC-Linz artificial neural networks interest group (ANNig).

In 1990 the ANNig has been accepted as associated member of a branch of the big European initiative JESSI, based on a proposal made by J. Pfalzgraf. The basic idea of that proposal was to integrate two powerful simulation tools from robotics and neural networks, respectively, to establish a flexible, versatile simulation environment for the application of artificial neural networks (ANN) methods to robotics problems, mainly (inverse) kinematics.

The idea of an integrated simulation environment formed the starting point to look for the two basic components of such a simulation tool. Both were easily found: The robot/NC simulation tool SAVE is available at RISC in the Softautomation Lab. This software is an advanced product and allows to simulate specific robot arms. Although it was mainly designed for NC machines, it is general enough to model and simulate any tree of open kinematic chains. It can visualize the robot model using a standard graphics interface, PHIGS.

Concerning the ANN simulator we decided to take the system NETUSE, which is provided by our cooperation partner Dr. Hans Geiger (GNC) and Kratzer Automatisierung GmbH, Munich. This system excellently fits into the project because it is available in source code for the ANNig and is therefore open for adaptations. The simulator is based on existing notions introduced by Dr. Geiger, which allow to model general network topologies. One of the basic ideas in this approach is the local (geometric) structuring of the networks.

In the course of these fruitful cooperations, the idea ripened to devise a completely new design of the ANN simulator. This was naturally motivated by obvious drawbacks and disadvantages which were identified in the practical use of the old version of the simulator. The initiative found great acceptance from the side of our partners. This new situation was the reason why the main part of the work then shifted to the proposal of a new concept for an ANN simulation system rather then concentrating on the integrated simulation tool although this latter "project" is still of central interest. Actually, in the last part of this work we discuss a prototype application of two cooperating simulation systems, namely SAVE and NETUSE, in a concrete example to demonstrate the principles.

The new work we are presenting in this thesis is a completely new design of a general command interface suitable for various purposes. Instead of a simple command dispatching mechanism, a full language is proposed. This language features all control constructs known from programming languages (like loops, selections, assignments, sub-routines, expressions, functions), strictly typed variables as well as language constructs to work in a restricted, object oriented fashion. The language has a programming interface to the language C, so that it can be extended with functionality to fit special purpose applications.

Following this, a language extension is specified which allows to generate artificial neural networks and to work with them. For this purpose, the object oriented features of the language are exploited. This strongly supports the reusability of already written code (generic programming techniques). This part of work definitely forms the new ANN simulator. Our concept design is flexible enough to allow the definition of natural interfaces to specific symbolic computation tools.

The thesis is organized as follows: Chapter 1 gives motivations and background information, especially from ANN and robotics. Chapter 2 describes the old version of NETUSE and identifies those drawbacks which motivated our new approach. Chapter 3 briefly describes the robot/NC simulator SAVE. Chapter 4 proposes the new command interpreter which eliminates the drawbacks found in NETUSE and introduces new features, most importantly an object oriented approach using instantiation and methods. Then Chapter 5 describes, how this new command language can be applied to establish our artificial neural network simulation system, in particular for simulating the network paradigm invented by Dr. Geiger, which was already implemented in NETUSE. In Chapter 6 we briefly come back to the original project idea, the integrated simulation tool. We describe a prototypical implementation to demonstrate the principle of such an integrated environment. Finally, Chapter 7 gives concluding remarks and some prospects of future work.

Acknowledgements. First of all, let me express thanks to Prof. Bruno Buchberger for the possibility to do this work in the stimulating and active environment of the RISC institute and for introducing me to the science of scientific work—thinking (logic and mathematical theorem proving), speaking (giving talks), writing (conducting papers).

Special thanks deserves my advisor, Dr. Jochen Pfalzgraf, who gave me the opportunity to write the work under his guidance, helping me with his experience and knowledge. He proved as an excellent, and patient, teacher, although his schedule was often very tight.

I thank also Dr. Hans Geiger for interesting discussions and suggestions in the field of his special network paradigm, and the NETUSE concept. The same holds for the members of the Kratzer Automatisierung group, especially Klaus Eder.

A "Thank you!" also goes to my colleagues Thomas Hainzl and Markus Meisinger for discussions and the friendly atmosphere in our ANNig-Lab.

Chapter 1

Introduction

In this section we present a brief selection of some motivational remarks for our work and background information about artificial neural networks (ANN) and robotics. For details we are referring to corresponding literature.

1.1 Motivation

The application of robots and robotic manipulators in industry has become standard today. The two main situations where robots do work are:

- Where the work is monotonous and would be too tedious for human workers. This is often the case with assembly-line work. This field of application is also optimal for robots because the procedure consists of repeating patterns.
- Where the environment is too dangerous for humans, such as in nuclear power plants, or at lacquered work or at places where humans cannot survive, for example in space or under water.

A robot must be programmed to perform a specific task. The wideliest used method for programming is known as “teach-in”, which is commonly used in car industry: For this method, the endeffector, for example a welding device, is dragged along the path which it must run later. The control software registers the positions at the joints in short time intervals (milliseconds), so that the path can be replicated.

This method is tedious, time-consuming, and error-prone. A more accurate method is desirable. However, the currently available methods for accurate control of robotic manipulators are hardly to handle computationally, are often numerically unstable and mostly work only for specific manipulators with certain special conditions concerning the axes of the joints. Furthermore, since the machine to be programmed is not available for normal work during programming, this period of time costs money, but is unproductive.

The control of robotic manipulators turns out to be an extremely hard problem. Even for fairly simple robots the kinematic equations become large, if only the kinematic aspects are considered (positions, velocities) but all forces (loads inertia) are ignored. If these are also included in the calculations the analytic approaches quickly become intractable.

For these reasons—intractability of problems in robot control and high costs during programming—, a different approach is tried: The real robot is replaced by a software simulation; the conventional methods for control are replaced by learning data structures: artificial neural networks.

Today, there is a growing community of researchers who try to apply connectionist methods (artificial neural networks, ANNs) to robotic control problems. There exist numerous different models how the arising problems can (at least partially) be solved.

This thesis intends to build the basics for a software tool that allows to study the connectionist approach to treat robotic problems systematically. To reach this goal an existing robot simulator and an artificial neural network simulator are integrated into one system. The ANN simulator is general enough to allow many different ANN paradigms to be implemented. The robot simulator is also a general tool which can model any tree of open kinematic chains. It provides forward kinematics calculation of the modeled robot arm, collision checking, and visualization. More details about the simulation tool follow subsequently.

1.2 Neural Networks

1.2.1 Theoretical Aspects of Neural Networks

First research in connectionism has already begun several decades ago, when the first theories in neuro-science were formulated. In [Heb49] D. O. Hebb reasons for his postulate that the fiber connecting two simultaneously firing neurons "thickens". A reprint of the Introduction and Chapter 4 appears as [Heb88]. His researches lead to an important unsupervised learning method, the "Hebb learning rule".

[Gro88] gives a historical review on the research in artificial neural networks as well as a comprehensive and mathematically oriented survey on well-known and less well-known network paradigms and learning methods.

[Sim90] provides a comprehensive survey of paradigms of artificial neural networks which have been investigated up to now. One of the most important types of ANN, the Perceptron, was invented by Frank Rosenblatt. In [Ros58] he gives a formal proof of convergence of the training method.

The basis for the backpropagation network, definitely the most popular paradigm, was founded by the invention of the Delta Rule, also named after its inventors Widrow-Hoff-Rule [WH60]. The Generalized Delta Rule (see [RHW86]) allowed to train hidden layers in networks, which was an unsolved problem so far.

Another good source of information is the textbook [Roj93], which treats the most important ANN paradigms theoretically and also discusses hybrid models.

There is also a trend to combine fuzzy logic methods with artificial neural networks and to study the integration of ANN and genetic algorithms. ([Hai94] presents an overview on this topic.)

1.2.2 What is an ANN?

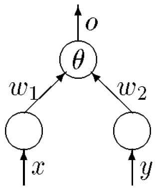
For the convenience of the reader we give a short summary of the basic constituents, which describe an ANN.

In general, an ANN consists of artificial *neurons* (or *processing elements*, *PEs*). These are coupled by (directed) *connections* (also called *synapses*) following a certain network topology. Neurons can be grouped together in layers. If a network is layered, there usually exist specific layers: an *input layer*, at which the network is fed with data; an *output layer*, at which the result after a so-called *recall step* is provided, and possibly *hidden layers* between them. There are usually no connections among neurons of a layer but only between neurons of different layers. The connections are usually weighted (*connection strength*).

The "standard" components of a neuron as we are using it are: a current state (*activation*), a transfer function, an output function. The calculation of the activation of a neuron is described below. The output of a specific neuron can be part of the input to other neurons. The input to a neuron is commonly the sum of the weighted output of all neurons from which it gets input. This input is transformed by the transfer function, whose result is the new activation of the neuron. The transfer function sometimes also takes into account the old activation of the neuron. The output of the neuron is computed from the activation by the output function. However, the output function is commonly only the identity function; hence, the activation is identical to the output.

The remainder of this section describes an important benchmark problem, the XOR-problem, which arises in various disguises in the field of artificial neural networks.

The XOR-problem



Starting point is the following question: Is it possible to express the logical And, Or, and Xor functions by an artificial neural net without hidden units? What we want to represent is a function of the type $\{0,1\} \times \{0,1\} \to \{0,1\}$, consequently the net has two inputs and one output units. The output unit shall be a threshold unit (cf. the picture to the right). The input to the output neuron is

$$A = w_1 x + w_2 y, \tag{1.1}$$

where x and y are the inputs to the net and w_1 and w_2 are the weights of the connections from the inputs to the output neuron. The inputs can only be 0 or 1 for both x and y. The net output o is

$$o = \begin{cases} 0, & \text{if } A < \theta, \\ 1, & \text{if } A \geq \theta, \end{cases} \tag{1.2}$$

where θ is the threshold of the output neuron. w_1, w_2, and θ are the quantities which the net shall find by learning. We want the output o to conform to the table

x	y	And	Or	Xor
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0
1	1	1	1	1

It is easy to verify that for example $(w_1, w_2, \theta) = (1, 1, 1.5)$ represents the And function and $(1, 1, 0.5)$ the Or function. Let us investigate whether our goal for the Xor function is achievable *in principle*. For the output o to be 1 the inequality

$$w_1 x + w_2 y \geq \theta \tag{1.3}$$

has to be fulfilled (cf. Eq. (1.1) and Eq. (1.2)). We can now interpret the pairs (x, y) as points in the plane. Eq. (1.3) represents the area on one side (and including) the straight line with equation $w_1 x + w_2 y = \theta$. The process of training the net is equivalent to finding the parameters w_1 and w_2 of a straight line such that all input points whose corresponding output shall be 1 lie on one side of the line and all input points whose corresponding output

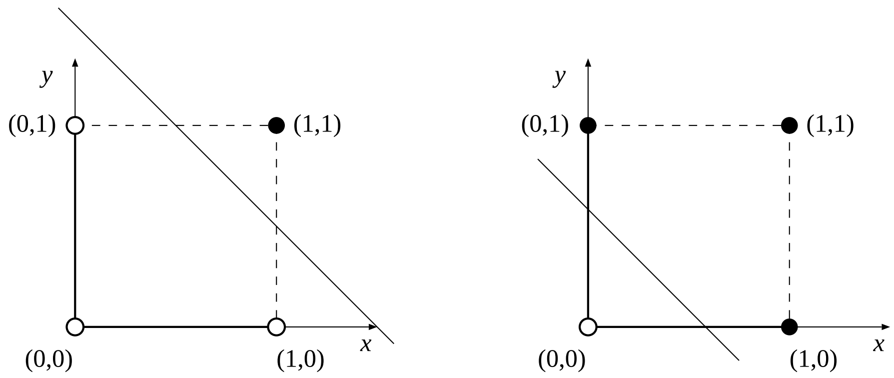


Figure 1.1: Separating lines for the And-problem (left) and the Or-problem (right).

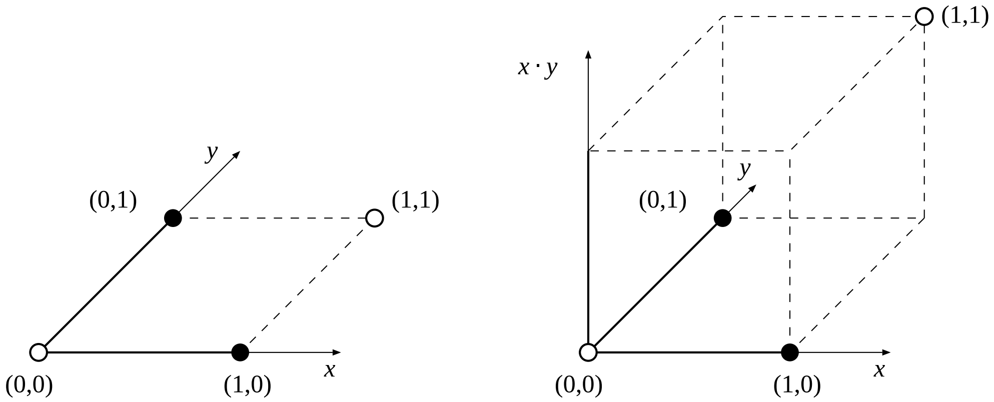


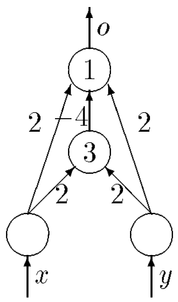
Figure 1.2: Distribution of the values of the Xor-function: Original two-inputs-net (left) and the net with functional link $x \cdot y$ (right).

shall be 0 lie on the other side of the line. Indeed, as Fig. 1.1 illustrates, for the And-problem and the Or-problem such lines can easily be found. These input sets are said to be *linearly separable*. However, for the Xor-problem there does not exist a separating line (see Fig. 1.2, left).

In [Pao89] *functional link* nets are introduced. The idea is to increase the dimensionality of the input by adding L units $\mathbf{i}_i$ (with $1 \leq i \leq L$) to the input layer of the net which receive, in general, the input

$$\mathbf{i}_i = f_i(\mathbf{x}_1, \ldots, \mathbf{x}_n),$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are the original inputs.



Reconsidering the Xor-problem, let us now add a third input unit to the existing network, which receives the input $x \cdot y$. Extending Eq. (1.3) to three dimensions, we immediately see from Fig. 1.2 that we can find a plane which separates the inputs which yield output 0 from those which give output 1. Alternatively, Xor can be represented by introducing one hidden unit in the original net (see picture to the right). This motivates excellently the introduction of hidden layers in neural networks. A basic problem which arises then is how to train hidden units. Historically, this was an important problem ultimately solved by the backpropagation algorithm [RHW86].

1.2.3 Remarks on Selected Simulators

Up to now, many ANN simulators have been implemented, which often can operate several different kinds of network paradigms and learning methods. For doing research, however, it is usually necessary for the scientist to have the possibility to extend and modify the algorithms in use. Therefore, a "closed" system is not desirable. Since commercially available simulators are usually "closed" (in the sense that the customer cannot change the program), it is not surprising that nearly each ANN-research institute implements its own simulator. In the following, a few more or less "open" systems are mentioned.

Probably the wideliest used ANN simulator currently is Neural Works Professional II by NeuralWare, Inc. It is a commercial product available for many different hardware platforms and operating systems, including PCs, Sun workstations, and Macintosh computers. The system includes many paradigms [Neu88] and a language is available, with which new methods of learning can be designed. The simulator has a graphical user interface, which allows to generate a network and to operate it. Different kinds of "instruments" are available to watch the performance of the network. Although the graphical interface allows to operate on the network in an easy fashion, it imposes also restrictions on the size of networks which can be worked with. For this reason the simulator can best be used for prototyping and research, but professional solutions are only possible if the networks are small and simple. But this tool does not allow to manipulate the source code of a prepared net, it is not an "open" system in the above sense.

Nearly the counterpart of Neural Works is NETUSE. This simulator was developed at Kratzer Automatisierung Gmbh, Unterschleißheim (Munich). It runs on Unix systems only and has a command line oriented user interface, but it is possible to construct large networks with thousands of neurons and connections. It has already been applied successfully in industry for pattern recognition, quality control, and process control. The simulator is used as basis for this thesis and is more thoroughly described in Chapter 2.

Another simulator was implemented at the University of Stuttgart, the SNNS (Stuttgart Neural Network Simulator), which is available for free. It has a graphical user interface based on the X Window System, and currently only runs on Unix workstations. The SNNS implements, among others, the back propagation network and allows to study it using different kinds of "instruments".

A further commercial implementation of an ANN simulator is the Nestor Learning System by Leon Cooper and Charles Elbaum (Nestor, Inc). It is based on the *Reduced Coulomb Energy Method* and specializes on pattern and character recognition.

1.3 Robotics

1.3.1 Theoretical Aspects

The Robot Map

In this section we make some motivational remarks showing how a mathematical model for a robot can be established. This *robot map* in its general form involves well-known mathematical manifolds—this is the reason why this modeling approach opens the possibility to apply various results and methods from other disciplines, especially geometry and (algebraic) topology, in a very natural way [Pfa93].

Basics. A robot (*robot arm* as a whole) as we are considering it can be briefly described as a system of rigid parts (*sub-arms*, *links*) where successive, neighboring links are connected by *joints* and which in many cases is an open kinematic chain with a tree-like structure.

The description of the motion of such a system is studied in the classical field of kinematics and geometry (in classical mechanics as a branch of physics and engineering). In kinematics the motion of rigid bodies is studied without considering forces and momentums—these are considered in robot dynamics. Hence robot kinematics is dealing with the study of the (time dependent) curves of motion, velocities (first derivatives of the curve of motion) and higher derivatives, if necessary, but thereby neglecting forces and momentums.

Of central importance for these considerations is a fundamental theorem of Leonard Euler (1776), see e.g. [Ang88], stating that the motion of a rigid body where a point remains fixed is a rotation around an axis (being fixed) going through that point.

As already mentioned, a robot model will be described as a kinematic chain (mostly open with tree like structure) consisting of *links* (sub-arms) and *joints*. Two neighboring links are connected by a joint, where motion takes place. Technically the most common types of joints are *rotational* (*revolute*) and *translational* (*prismatic*) joints. The *base* of a robot is that part where a robot is fixed to the ground or where the wheels are attached, for example. The first sub-arm is then mounted at the base. The last sub-arm of the whole robot arm is usually used for attaching tools like screw drivers, welding devices, etc.; it is called the *end effector* (EE for short).

Notions. The *joint space* (also called *configuration space*) of a kinematic chain is a set of vectors, where each component of a vector specifies a parameter value for a single joint in the kinematic chain [LP80]. The value specifies an angle if the joint is rotational or a distance if the joint is translational.

The *work space* is the set of vectors which specify the position and orientation of the end effector. Details follow below.

The robot map. Let t denote the number of translational joints and r the number of rotational joints. The objective is to model the robot as a mathematical map from joint space $\mathcal{C}$ to work space $\mathcal{W}$ (cf. e.g. [Kar88], [Bur89], and also [DPSS91]):

$$\mathsf{R} : \mathcal{C} \to \mathcal{W}$$

The *degree of freedom* (DOF) is defined to be $\mathrm{DOF} = n = r + t$ (this is the total number of joints $n = r + t$), i.e. the "joint space manifold" $\mathcal{C}$ has dimension $r + t$.

The question arises about the largest possible joint space (where all smaller joint spaces can be embedded). To this end we consider the following.

The parameter of a translational joint runs through a real interval, we model this more generally by the whole real line $\mathbb{R}$. Rotational joint parameters can analogously be interpreted as elements of the unit circle S^1 in the real plane; recall that $S^1 = \{z = \mathrm{e}^{\mathrm{i}\phi} \mid 0 \le \phi < 2\pi\}$ (for the algebraic description we prefer here the complex number representation of points of the real plane) and a rotation through angle ϕ corresponds to a multiplication of a 2-dimensional vector by $\mathrm{e}^{\mathrm{i}\phi}$ in the corresponding "rotation plane" (the complex numbers represented as 2-dimensional vectors with real and imaginary parts as x- and y-components in the real plane; recall that $\mathrm{e}^{\mathrm{i}\phi} = \cos\phi + \mathrm{i}\sin\phi$). Thus the configuration space (joint space) manifold $\mathcal{C}$ can in the general sense be represented by

$$\underbrace{(\mathbb{R} \times \cdots \times \mathbb{R})}_{t} \times \underbrace{(S^1 \times \cdots \times S^1)}_{r} = \mathbb{R}^t \times (S^1)^r,$$

where the first Cartesian product has t factors, and the second r factors. The Cartesian product $(S^1)^r$ is also called *r-dimensional torus* $\mathbb{T}^r$ with $\mathbb{T} = S^1$, the 1-dimensional torus.

From a mathematical point of view this big configuration space is a nice differentiable manifold, which allows that results from analysis on manifolds can easily be applied for studying the configuration space.

The corresponding question concerning the *work space* is: What is the complete information necessary to describe the location of the end effector of a robot? The complete information about the EE movement is described by a *position vector* $a = (a_1, a_2, a_3) \in \mathbb{R}^3$ and the *orientation* of the EE.

Orientation in Euclidean 3-space can be uniquely determined by an orthonormal frame. The variety of all frames can be described by the group of all proper rotations, the *special orthogonal group*: $\mathrm{SO}_3(\mathbb{R}) = \{A \mid A$ is an orthogonal 3×3 matrix with $\det A = +1\}$.

Hence the whole work space manifold, describing all possible EE locations in the most general case, is $\mathbb{R}^3 \times \mathrm{SO}_3(\mathbb{R})$, the *Euclidean motion group*. Again, this is a nice differentiable manifold. It is of dimension 6, this is why 6-DOF robots are so widespread in practical applications. More precisely, a manipulator with the capability that the EE can reach every point in the complete work space must at least have six DOF, i.e. it needs six joints and at least three of them have to be rotational (to provide all possible orientations).

Thus, in the most general case, a robot can be mathematically described by a (differentiable) map, the *robot map*.

$$\mathsf{R} : \mathbb{R}^t \times (S^1)^r \to \mathbb{R}^3 \times \mathrm{SO}_3(\mathbb{R}),$$

This is the general robot map which is used for theoretical studies. We mention here that, of course, the mechanical structure of a robot determines its workspace structure (geometry and topology). Technically relevant robot structures frequently have cube, cylinder, ball, or torus-like workspace structures. These special joint and work spaces, respectively, are then mathematically embedded in the above large spaces.

Using the previously introduced notation, let $\mathsf{R} : \mathcal{C} \to \mathcal{W}$, in general, denote the robot map describing a selected robot model with joint space $\mathcal{C}$ and work space $\mathcal{W}$. Let $q \in \mathcal{C}$ denote the joint coordinate vector (n-dimensional) and $x \in \mathcal{W}$ the corresponding coordinate vector (m-dimensional) in the work space, respectively. With this we define the following basic notions.

As already mentioned earlier, the degree of freedom (DOF) of the manipulator system is defined by the dimension of the configuration space manifold, i.e. $\mathrm{DOF} = \dim(\mathcal{C}) = n$.

A *redundant manipulator* is defined as a robot having more DOF than is necessary in comparison with the dimension of the workspace, i.e. $\dim \mathcal{C} = n > m = \dim \mathcal{W}$.

Redundant manipulators are of increasing interest for complex industrial applications where difficult EE movements have to be performed and where redundancy is exploited practically. From an analytical point of view, redundancy increases the mathematical complexity of the problem (difficulties concerning the analytical treatment of kinematics and dynamics).

Problem formulation. Subsequently we formulate the two typical problems in robot kinematics using the robot map (namely the direct or forward kinematics problem and the inverse kinematics problem, respectively).

Let $x = (a, A) \in \mathcal{W}$ denote a point in work space specifying an EE location and $q = (q_1, \ldots, q_n) \in \mathcal{C}$ a point in the configuration space with joint coordinates (parameters) $q_1, \ldots, q_n$.

Forward (direct) kinematics problem: Given a joint configuration q, determine the EE location corresponding to these joint parameter values. That means in terms of the robot map for the given input q determine $\mathsf{R}(q) = x$.

Inverse kinematics problem: Given a point $x = (a, A)$ which specifies an EE location then determine a suitable joint configuration $q = (q_1, \ldots, q_n)$ such that the robot points to the given EE location x. In terms of the robot map: Given x, find a suitable q (pre-image) such that $x = \mathsf{R}(q)$.

More general, the inverse kinematics problem deals with the description of the pre-image manifold $\mathsf{R}^{-1}(x) = \{q \in \mathcal{C} \mid x = \mathsf{R}(q)\}$ for a given point x of the work space (cf. e.g. [Bur89]) and the task to select (in a certain optimal way) a point q in the joint space that belongs to $\mathsf{R}^{-1}(x)$, i.e. $x = \mathsf{R}(q)$, in the case that $\mathsf{R}^{-1}(x)$ contains more than one element.

In robot control the inverse kinematics problem is (among others) treated in its first order linearization (velocity control) involving the derivative (Jacobian) of the robot map R (here eventually the pseudo inverse of a matrix is a useful tool).

As mentioned in [Wlo92] the inverse kinematics tasks which arise in robotics applications concerning the currently used industrial robots are solved using numerical methods. But besides that it is desirable to have closed form symbolic solutions available. [Hei85] solves the kinematic equations for a class of robots.

Specification of Coordinate Transformations

Workspace coordinates. We will now inspect the specification of work space coordinates $x \in \mathcal{W} = \mathbb{R}^3 \times \mathrm{SO}_3(\mathbb{R})$ in practice. The position part of a work space coordinate vector is a 3-dimensional vector $a \in \mathbb{R}^3$ (a 3×1-matrix). The orientation part $A \in \mathrm{SO}_3(\mathbb{R})$ is a 3×3-matrix. For technical reasons we combine now the orientation matrix and the position vector to form the 3×4-matrix

$$\left(\, A \mid a \, \right).$$

Transformations. From the possible coordinate transformations (like translation, rotation, stretching, perspective projection) we will only consider translations and rotations.

A translation of a coordinate frame is achieved by adding the translation vector $t \in \mathbb{R}^3$ to the position part of the coordinate frame:

$$\left(\, A \mid t + a \, \right).$$

A coordinate frame is rotated by $R \in \mathrm{SO}_3(\mathbb{R})$ by multiplying both the orientation part and the position part of the coordinate frame by R:

$$\left(\, R \cdot A \mid R \cdot a \, \right).$$

We can use a trick to simplify and unify translation and rotation to one simple operation: matrix multiplication. The coordinate frame is augmented by the row vector

$\left(\begin{array}{c|c} \underline{0} & 1 \end{array} \right)$, where $\underline{0}$ is a 1×3-null vector:

$$\left(\begin{array}{c|c} A & a \\ \hline \underline{0} & 1 \end{array} \right).$$

The resulting frame is a 4×4-matrix. This specification of a coordinate frame is known as *homogeneous coordinates* [Pau81, Chapter 1].

We now specify a translation by t as

$$\left(\begin{array}{c|c} I & t \\ \hline \underline{0} & 1 \end{array} \right), \quad \left(\begin{array}{c|c} I & t \\ \hline \underline{0} & 1 \end{array} \right) \cdot \left(\begin{array}{c|c} A & a \\ \hline \underline{0} & 1 \end{array} \right) = \left(\begin{array}{c|c} A & t+a \\ \hline \underline{0} & 1 \end{array} \right),$$

where I is the 3×3-identity matrix.

A rotation of R is specified as

$$\left(\begin{array}{c|c} R & \vec{0} \\ \hline \underline{0} & 1 \end{array} \right), \quad \left(\begin{array}{c|c} R & \vec{0} \\ \hline \underline{0} & 1 \end{array} \right) \cdot \left(\begin{array}{c|c} A & a \\ \hline \underline{0} & 1 \end{array} \right) = \left(\begin{array}{c|c} R \cdot A & R \cdot a \\ \hline \underline{0} & 1 \end{array} \right),$$

where $\vec{0}$ is a 3×1-null vector.

From a Robot Arm to its Robot Map

We will now briefly sketch the problem of finding a mathematical model from a physical robot arm. [Sixt] provides a tutorial for this task; cf. also [Pfa93] for a discussion of it.

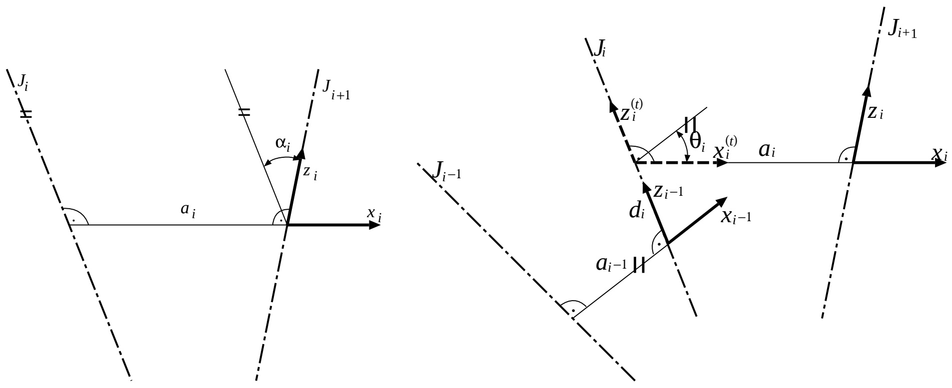
The goal is that the end effector's position and orientation are expressed with respect to a world coordinate system in a way which involves the local contributions of each link and joint position.

Given is a robot arm whose technical data (lengths of links, angles if links are not straight, kinds of joints, etc.) is well known. We want to find a mathematical description, in particular the robot map R of the robot arm. The robot map is a function which depends on n variables, which is the number of joints in the robot arm (DOF). The difficulty is that for a general arm the technical data is not provided in a way such that the robot map can be devised directly.

In [DH55] a method is invented which allows to find the robot map from the technical layout of a robot arm. The method leaves the joint parameters as variables in the robot map during its establishment as desired. [Pau81] further described the methods and also introduces some shortcuts. However, as is outlined in [Sixt], these "simplifications" corrupt the natural intention of the method and complicate things.

For this method of finding the *Denavit-Hartenberg parameters*, the links and joints are numbered in sequence, where the base is link L_0 and the endeffector is link L_n (n is the number of joints in the kinematic chain). The joint towards the base of link L_i is numbered J_i and towards the endeffector J_{i+1}.

Now coordinate systems are assigned to each link. In the same step the *link values* of the link can be derived. See Fig. 1.3(a). The origin of C_i, the coordinate system of link L_i, is assigned at the intersection of joint axis J_i and the common normal between J_i and J_{i+1}. The length of this common normal is the *link length* a_i. The z_i-axis coincides with the joint axis J_{i+1}. The x_i-axis coincides with the common normal. In general, the orientations can be chosen arbitrarily, but they will usually "point towards the endeffector". Lastly, the y_i-axis will be chosen such that the coordinate system C_i is a "right-handed" system.



(a) (b)

Figure 1.3: (a) Link i, (b) Joint i.

The *link twist* α_i is the angle between a straight line through the origin of C_i parallel to joint axis J_i and joint axis J_{i+1}.

Fig. 1.3(b) illustrates how two further values, the *joint values*, are derived. For this, two consecutive links L_{i-1} and L_i are considered. A temporary coordinate system $C_I^{(t)}$ is introduced. Its origin is at the intersection of J_i and a_i. The $z_i^{(t)}$-axis coincides with the z_{i-1} axis and the $x_i^{(t)}$-axis coincides with the x_i-axis. The distance between the two common normals a_{i-1} and a_i on the joint J_i is the joint value d_i. The angle between these two common normals (in a plane perpendicular to J_i) is the joint angle θ_i.

The values a_i, α_i, d_i, and θ_i ($1 \leq i \leq n$) are the Denavit-Hartenberg parameters of the kinematic chain. a_i and α_i are always constant because they describe features of a link, which is rigid. If joint J_i is a revolute joint, the angle θ_i is the joint parameter and d_i is constant. Otherwise, if J_i is prismatic, d_i is the joint parameter and θ_i is constant.

During the determination of Denavit-Hartenberg parameters special situations can arise, such as parallel joint axes, coinciding or intersecting joint axes. Such situations usually occur in industrial robot arms because they drastically simplify the robot map and, therefore, the mathematical treatment. Although consideration of these situations is important, we refer to [Sixt] for details at this point.

1.3.2 Some Remarks about Robot Simulation Systems

Numerous robot/NC simulators have been implemented up to now. An equal number of further systems have only reached the planning phase or have been stopped implementation. [May88] gives a comprehensive survey on existing simulators in industrial use as well as on systems which have been planned only.

Most of the systems are implemented in FORTRAN, some in C and Pascal. Many systems are extremely “optimized” for special graphics hardware to gain speed. However, these systems are not portable for this very reason. Only few systems are general enough to refrain from such “optimizations” and support a modern graphics standard like PHIGS. Similarly, only few systems produce a standard robot control code.

When looking for a “universal” simulator which combines modeling, simulation, programming, usage of standards for object and task input as well as for both graphic output and robot code, only two systems remain: ROSI [DH86] and SMART [MHÖ89] the ancestor of the SAVE system, which is described in Chapter 3.

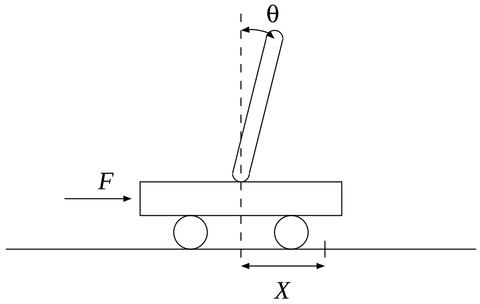


Figure 1.4: The pole-cart problem.

1.4 Discussion of Some ANN Approaches

1.4.1 Some Problems in Robotics Solved with ANN

Broom-Balancing Problem

The *broom-balancing problem*, which is also known as the *inverted pendulum* or the *pole-cart problem*, is a classical example of applying artificial neural networks to a control problem. Given is an inverted pendulum of mass m with length L mounted on a cart with mass M. The goal is to move the cart by applying a force F in horizontal direction such that the pendulum keeps balance (i.e. angle $\theta = 0$) and the cart remains at its origin ($X = 0$) (see Fig. 1.4). The equations of motion are:

$$\begin{aligned}\ddot{\theta} &= \frac{3}{4L}(g \sin\theta - \ddot{X}\cos\theta) \\ \ddot{X} &= \frac{m(L\sin\theta\dot{\theta}^2 - \frac{3}{8}g\sin 2\theta) - F\dot{X} + u}{M + m(1 - \frac{3}{4}\cos^2\theta)}\end{aligned}$$

This is an undamped and inherently unstable fourth-order system whose dynamics are nonlinear and coupled. ([HJV90] states the above formulas.)

This problem has already been considered in [WS63]. [CP89] presents a solution for this problem using ANN.

Inverse Kinematics

Although an exact solution to the inverse kinematics problem is desirable it is often expensive to get.

Let the control of a human arm serve as a model. Of course it is not controlled by inverse kinematics. A two phase controlling scheme is applied instead: In the first phase the arm is moved rapidly to bring the hand in the vicinity of the object to handle. This phase employs usually only *feed-forward*: I do not need to look at the object to move my hand near the object. Yet to grab it I look at the object and my hand. This is the second phase where *sensor feedback* is used for fine-controlling the hand.

In [YG89] the concept of *context sensitive networks* is used for learning inverse kinematics. These networks introduce different levels of abstraction (and hence semantics). The idea is that the input to the network is partitioned into a context input and a function input. The context input goes into a context network whose output controls the weights of a function network which computes the overall output when receiving the function input.

In [RMS89b] an extension of Kohonen's self-organizing mapping algorithm is used to learn the visuo-motor-coordination of a simulated three-link robot arm. This algorithm is capable of learning a mapping between a sensory input space and a motor output space by establishing a topology-conserving map on a planar array of neural units. Two weight vectors are maintained by each neuron, one for the input and one for the output. During learning both vectors are adjusted. Each neuron represents a (connected) subset of the input space. For the adaptation of the output vector an error correction method of the Widrow-Hoff-type is employed ([WH60]). A Gaussian-shaped function is used to calculate the adaptation terms. Upon recall the output-vector associated with the winning neuron is delivered as output. [RMS89a] provides a program in PASCAL which implements these ideas.

Inverse Dynamics: Manipulator Control

The paper [RMS89b] also shows the application of an extension of Kohonen's self-organizing mapping algorithm to the computation of inverse dynamics. The integration of the dynamic system, however, seems not to be done carefully since the authors put the centers of gravity of the manipulator links into the centers of the link frames which, in general, do not coincide with the centers of gravity.

[Kup91] introduces INFANT, a neural controller. This controller learns three tasks:

1. Locating stationary targets with movable sensors.
2. Reaching arbitrarily positioned and oriented targets in 3D space with multi-joint arms.
3. Positioning an unforeseen pay-load with accurate and stable movements despite unknown sensor feedback delay.

The controller is mainly based on a linear network with a fairly simple learning rule.

In [MKSS88] a learning method is described which requires neither an exact model nor parameter estimation. The method generates motor commands directly from the "goal of the movement".

1.4.2 Some Remarks on other ANN Application Fields

Artificial neural networks are already successfully applied in pattern recognition, speech recognition, non-destructive testing, noise filtering, data compression. ANN are also capable of doing complex control tasks. A broad field for applications is aeronautics and space navigation, where error tolerant and adaptive systems are necessary. Recently, ANN are also surprisingly often successfully applied in banking.

[SHG90] applies different neural networks to stock price prediction. The authors apply an Adaline network, a Madaline network, and a backpropagation network to three different stocks. The networks were able to predict correctly whether the prices will rise or fall at rates of 60% to 75%. These rates, although they are seemingly weak, are significantly better than "random tips". The backpropagation network could also predict the prices fairly well. All in all, it is difficult to predict stock prices because they do not only depend of concrete factors (such as company profits) but also on psychological factors like rumors or political situations, which can hardly be suitably represented for electronic processing.

Chapter 2

ANN Simulator NETUSE

The artificial neural network simulator NETUSE was mainly designed to implement an ANN paradigm which goes back to ideas and results of Dr. Hans Geiger, who started to work in the field several years ago. The implementation was done by Thomas Waschulzik at Kratzer Automatisierung Gmbh, Munich, under the guidance of Dr. Hans Geiger (who formerly was the R&D-leader of the company, but recently opened his own company GNC). It is described in the diploma thesis [Was87].

A basic principle in Geiger's network paradigm is to model an ANN which is locally regularly structured. This "local-global" modeling principle has advantages over those paradigms in which the net is globally controlled by certain cost functions (like in back-propagation nets). For a thorough discussion of this network and its mathematical model see [GP94].

In this chapter, the version 4.0 of NETUSE is described.

2.1 The Concept

A major request which the designers of NETUSE wanted to carry out was a great flexibility in the network design, in particular the construction of complex network topologies such as network cascades and modularized design. NETUSE is a system of several programs. The two most important programs are:

- A compiler which translates the language NDL (*Network Description Language*). It generates a program which in turn builds a network when it is executed.
- The *Network Test System* which lets the operator perform actions on the network, such as reading the network, doing training and recall.

Several additional tools exist, one of which allows to merge two networks into one network.

Fig. 2.1 illustrates the process of generating a network. Obviously, this process is complicated. The reason for this is that the networks which can be built with netuse can have arbitrarily complicated topologies. Furthermore, practical networks tend to have thousands of neurons and connections. Such networks are usually intractable with a more user-friendly human-machine interface (e.g. a graphical interface). So it was necessary to find some means which allows to construct networks of that size and complexity. A description language fits best for this task.

The Network Test System is an interactive, command-line oriented tool. Since NETUSE was first developed on slow DEC-VAX machines running the operating system VMS with-

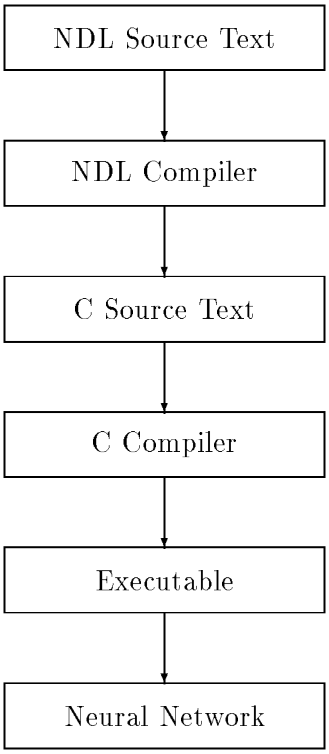


Figure 2.1: The process of generating a neural network for NETUSE.

out good graphical support, a command interpreter was the only choice the developers had.

2.1.1 NDL—Network Description Language

For NETUSE a language was developed which allows to describe an ANN and its parts. There are two different kinds of objects which must be described: *neurons* and *connections*. The following specifies a neuron:

Adjustment This part describes how a neuron behaves in a changing environment.

Plasticity specifies how the connections from which the neuron gets input are changed, i.e. how the network learns. Up to 32 numerical parameters can be specified whose interpretations are not fixed, but are left open. It depends on the learning method in the *Network Test System* how these values are interpreted. Different learning methods, in turn, are implemented in different commands in NTS.

Element type This declaration specifies a neuron type by assigning an adjustment, a plasticity, and a linear threshold function as in Fig. 2.2 with $a = 255$ constant. The threshold is applied to find the activity of the neuron, i.e. its output. No further output function is applied. For example, the type of an input neuron is determined by a null adjustment, a null plasticity (no input through connections is received, hence no learning is necessary), and a simple threshold because the network test system directly assigns the values of the input file to the activity of the neuron.

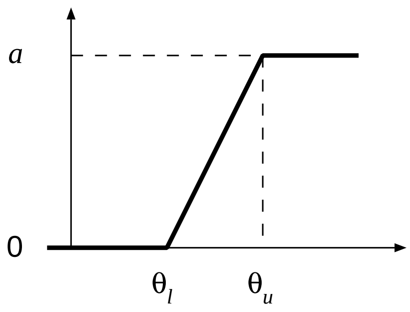


Figure 2.2: The linear threshold function.

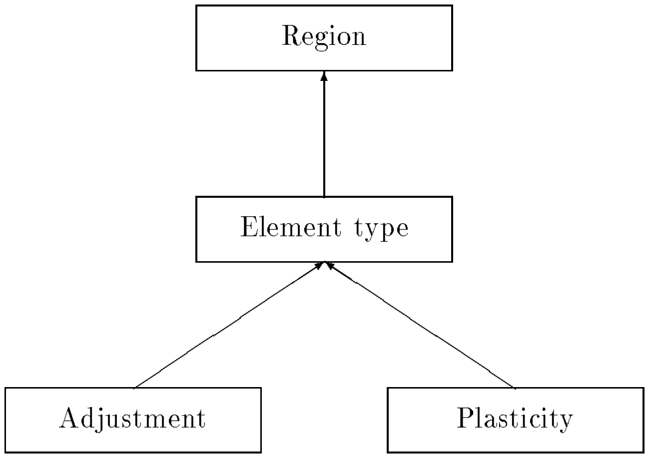


Figure 2.3: The hierarchy of the parts which specify the neurons in a network.

Region The region assigns element types to a range of neurons in the network. For example, in a two-layered network there is the region of input neurons and a region of output neurons.

Fig. 2.3 shows the dependencies among these parts. The arrow in the diagram means "is used by".

The following specifies the connections of the network:

Distance This function calculates the distance between two neurons. The result can later be used to initialize the connection strength between these two neurons. (This makes sense only if there exist connections among neurons in a single layer.)

Geometry The geometry defines how neurons in a certain range can be accessed. A dimension and a distance function can be specified. Currently, only 1, 2, or 3 dimensions are allowed, i.e. the neurons in the range can be viewed as arranged on a line segment, rectangle, or cube, respectively.

View The view assigns a geometry to a range of neurons. It is possible that different views overlap each other (partially or totally), providing different "points of views" over the same set of neurons.

Strength This defines a function which calculates the initial weight of the connection between two neurons. A distance function can be used to calculate the weight.

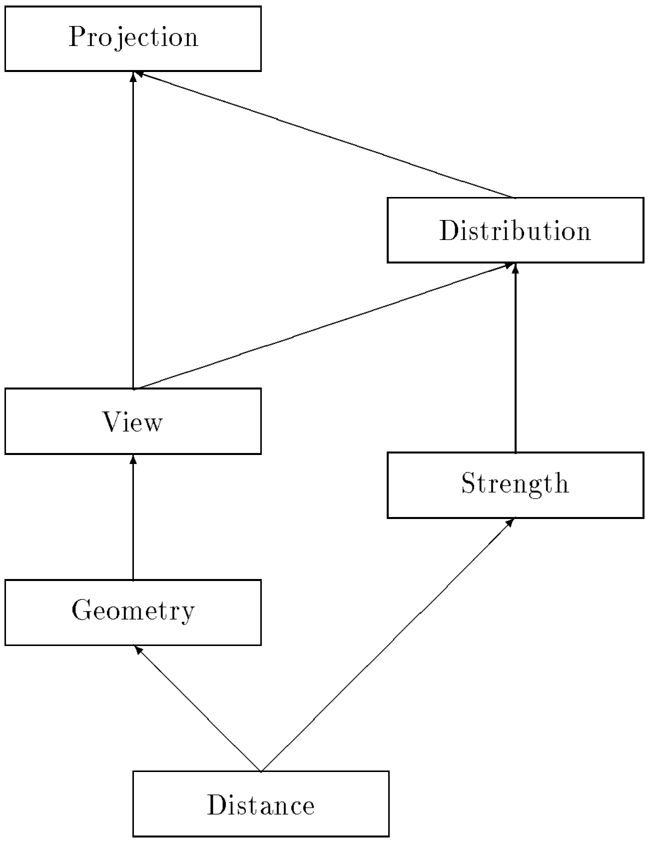


Figure 2.4: The hierarchy of the parts which specify the connections in a network.

Distribution The distribution selects the presynaptic neurons of a given postsynaptic neuron. This selection may be done with respect to an *origin*.

Projection The projection selects the range of the postsynaptic neurons. A distribution is specified to select the presynaptic neurons.

Fig. 2.4 shows the dependencies among these parts.

Usually the following holds:

- The view specifies all neurons of a single region.
- If the described network is layered, the origin of a distribution is often ignored because either all neurons are selected or an origin cannot be given because presynaptic and postsynaptic neurons come from different layers.
- For a layered network a projection for each layer must be specified.

Notice, that these are not restrictions imposed by the description language, but by the semantics of concrete networks.

2.1.2 NTS—Network Test System

The network test system is a program which operates on the network generated from the description. It handles

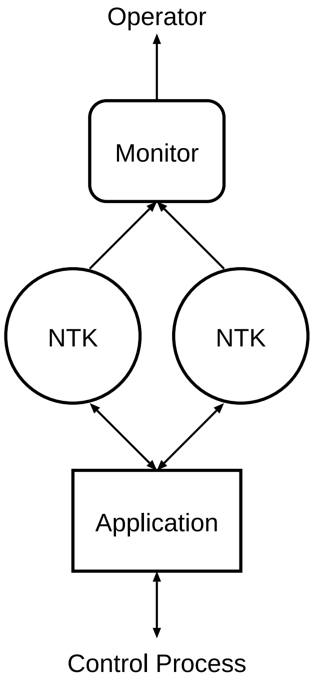


Figure 2.5: The client-server-model of the network test system.

- reading networks from and writing networks to files;
- reading network inputs from file;
- performing network executions;
- performing training of the network;
- viewing of network states.

Section 2.2 gives a detailed description of the internals of this program.

2.2 NTS—State of the Art

The network test system is implemented as a client-server-system. The *server*, called *NTK*, operates on the network. The client's task is to read the network from a file and to ship it to the server, provide input for the network, initiate operation on the network, and interpret its output. Many different clients can use servers.

Fig. 2.5 shows an example of an application structure. Let us assume that some control task shall be performed by a neural network. A program, the *application client*, uses 2 servers which operate on the network *in parallel*. The application receives input from sensors and sends this data to the servers (as network input) and receives network output from the servers which it interprets and performs the control task. A second client, the *monitor client*, also communicates with the same servers. Its task is to watch the performance of the network and to provide visualizations of the network outputs to an operator. The arrows show the flow of data. Notice, that the flow between the servers and the application client is bidirectional, whereas the flow between the servers and the monitor client is one-way.

The network test system consists of a kernel which is implemented as a library. The library provides the functions which read and write networks, data, and perform network actions. It is therefore easy to provide a variety of interfaces between the network and the real world. Some of the possibilities include:

- A program implements a fixed sequence of the function calls. This may be done in the application client in Fig. 2.5.
- An interpreter program provides a command line equivalent for each function and the operator determines the sequence of function calls by typing commands. This method may be used by the monitor client in Fig. 2.5.
- A graphical interface provides buttons and menu entries which execute the functions when selected.

The application client usually does not perform any training but only executes a network which has been trained offline using real-world data. Training is done using the monitor client. Training data is gathered using the capabilities of the application client. (For training, the data flow between the monitor client and the servers would also be bidirectional.)

2.3 The Command Language of the Monitor Client

The command interpreter included in version 4.0 of NETUSE is implemented as an independent module. The design goal was to make it independent from the program (the monitor client in the case of NETUSE) in which it is used. It supports the following constructs:

Internal commands These commands are intrinsic to the command interpreter. They include commands like `repeat`, `define`.

Variables The interpreter can maintain a list of named variables. A variable also maintains boundaries for the values it can take to help debugging.

Macros can be defined with the internal command `define` to simplify usage and to aid structuring of command files.

External commands These commands are supplied by the program which uses the command interpreter. They must be made known to the interpreter.

Here is an example session:

`NTS> const N_OUTPUT 5`	sets a constant
`NTS> getnet demo_5`	reads net from file `demo_5`
`NTS> n_mele?`	prints the number of connections
`        45`	
`NTS> n_vele?`	prints the number of neurons
`        14`	
`NTS> start_v=(n_vele-N_OUTPUT)`	sets a parameter value using an arithmetic expression
`NTS> end_v=n_vele`	sets another parameter
`NTS> activity=255`	sets yet another parameter
`NTS> chact`	changes activity of neurons `start_v` (included) to `end_v` (not included) to the value `activity`
`NTS> quit`	exit NTS

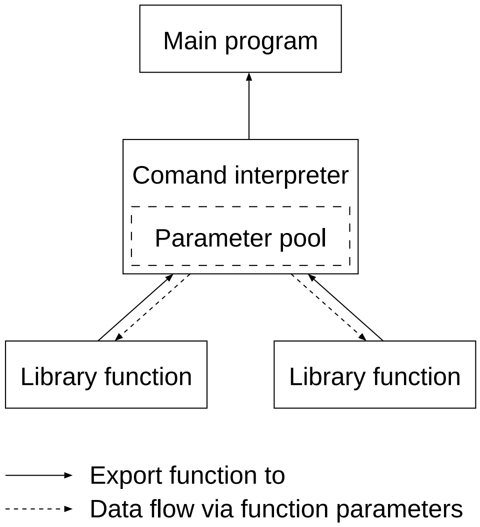


Figure 2.6: Function dependencies and data flow in version 4.0 of the monitor client.

The commands `const` and `quit` are internal commands. `getnet` and `chact` are external commands.

2.4 The Command Interpreter

Fig. 2.6 shows the dependencies among the modules from which the monitor client is built.

Main program The main program is simple. It performs some initializations. Then it invokes the command interpreter. When it returns, some cleanup is done and the monitor client terminates.

Command interpreter The command interpreter issues a prompt to the terminal and waits for a line of user input. Then it disassembles the line into its lexicographic parts and interprets them by invoking the library function corresponding to the command. Commands are made known to the interpreter by a static list, as we will see below.

Library functions The library functions operate on the network and the input data. Each function has a corresponding command in the command interpreter.

This section gives an outline how the command interpreter of version 4.0 is implemented.

2.4.1 Commands

The structure `costr` contains information for a single command:

```
typedef struct {
    char *strc;      /* Name of the command */
    char *helptxt;      /* String describing the command */
} costr;
```

The command interpreter allocates the variable `comdef`, an array of `costr` where the information for all commands is stored.

```
static costr comdef = {
        /* debugging */
        { "don", "Switch on debugging" },
        { "doff", "Switch off debugging" },
        /* commands related to reading and writing */
        { "getnet", "Read network from file" },
        { "putnet", "Write network to file" },
        /* commands operating on network */
        { "lsf", "Execute forced learning" },
        { "chact", "Change activity" },
        .
        .
        .
        };
```

After a command name is extracted from the input line the interpreter performs a linear search of this name in the array of available commands. The index of the entry is used to dispatch the library functions.

```
switch (index) {      /* dispatch commands */
case 0:      /* switch on debugging */
    debug = 1;
    break;
case 1:      /* switch off debugging */
    debug = 0;
    break;
case 2:      /* read a net */
    getnet(filename,      /* name of the file */
        &header,      /* header information */
        &nt_v,      /* vector of element types (neuron types) */
        &i_v,      /* vector of regions (index vector) */
        &v_m,      /* connection matrix */
        &n_vele,      /* number of neurons (vector elements) */
        &n_mele,      /* number of connections (matrix elements) */
        PNW_ASC);      /* file is in textual form */
case 3:      /* write a net */
    ...
    break;
case 4:      /* Execute forced learning */
    ...
    break;
case 5:
    chact(&i_v, start_v, end_v, activity);
    break;
.
.
```

.
}

2.4.2 Parameters

A similar approach is used for the parameters of the commands. The structure `parstr` contains information of a single parameter:

```
typedef struct {
    char *strp;        /* name of the parameter */
    int32 lolim;       /* lower bound of value */
    int32 hilim;       /* upper bound of value */
    char *helptxt;        /* Short string describing the parameter */
} parstr;
```

Version 4.0 of NETUSE supports only integer parameters in the range -2^{31} to $2^{31} - 1$. This range can be narrowed by the structure members `lolim` and `hilim`. The parameter descriptions are collected in the array variable `pardef`:

```
static parstr pardef[] = {
        { "n_vele", 0, 2147483647,
                        "number of neurons (vector elements)" },
        { "n_mele", 0, 2147483647,
                        "number of connections (matrix elements)" },
        { "start_v", 0, 2147483647,
                        "begin of element selection" },
        { "end_v", 0, 2147483647,
                        "end of element selection" },
        { "activity", 0, 255,
                        "new activity for chact" },
        .
        .
        .
        };
```

The parameter values themselves are stored in an additional array `param`:

```
#define MAX_PARAMS 300
static int32 param[MAX_PARAMS];       /* parameter values */
```

When the command interpreter encounters a parameter name it performs a linear search in `pardef`. The index at which the name is found corresponds to the index where the value of the parameter is stored in `param`.

In the `switch` statement where the commands are dispatched it is desirable to access the parameter by name and not by index. This is accomplished with the following `#define`'s:

```
#define n_vele param[0]
#define n_mele param[1]
#define start_v param[2]
#define end_v param[3]
```

```
#define activity param[4]
.
.
.
```

See the listing above for an example.

2.4.3 Discussion of this Approach

The implementation of the command interpreter in version 4.0 suffers from several drawbacks:

- The design goal that the command interpreter should be an independent module was not achieved. The command interpreter maintains a static list of variables and a static list of commands. If new commands are added the interpreter must be recompiled.
- If a new command is inserted in the existing list the whole switch statement which dispatches the commands must be rewritten to match the new numbers. Similarly, if a new parameter is inserted or an existing parameter is deleted, the whole list of defines must be rewritten to match the new indices of the parameters.
- The conditional command and the loop commands that are available are too restricted. They must be used in a way which deviates from the normal semantics of these commands in programming languages. A trick using separate command files must be used to get nested loops.
- The commands do not have parameters. Instead, the necessary "arguments" must be stored in global variables. Moreover, different commands share the same "parameter variables", which is extremely confusing for the user and a source of many errors which are hard to locate.
- The possibility to define macros is only an incomplete solution for the requirements which arise during practical work with the command interpreter, because the macros cannot take parameters. Of course, they do simplify the work with long and/or repeating command sequences, but a more flexible tool is desirable.
- The variables can have only one type of values, integers. In numerous applications, however, floating point numbers are desired. Also, strings would often be helpful.

To eliminate these flaws the whole system had to be redesigned. Chapter 4 proposes a replacement for the command interpreter which does not suffer from the above disadvantages. The new interpreter is a completely independent module. New commands and variables are added by calling utility functions and the details how variables and commands are maintained are hidden in the module. The control structures are more flexible and can be nested. Parametrizable functions and command procedures replace macro definitions, which increases the usefulness by orders of magnitude. Variables must be declared, but can have different types. Additionally, new features are introduced, most importantly semantics for restricted object oriented programming (instantiation, methods of object classes, structured types).

Chapter 3

Robot/NC Simulator SAVE

The Softautomation group of RISC-Linz designed and developed the robot/NC simulator SAVE (Simulation And Verification Environment). This was partly done in the frame of an industrial project with the company AIS (now AI Informatics) in Linz, Austria. [May91, Chapter 4] gives a comprehensive description. SAVE is designed to simulate NC machining processes. However, it is general enough to deal with arbitrary trees of open kinematic chains (in the robotic sense).

The main data structure used to represent a model of a robotic manipulator and its environment is the *layer tree* [MH90], which we will discuss in greater detail now.

To illustrate the concept we give an example. Fig. 3.1 shows a robotic manipulator of the Scara type. Fig. 3.2 shows the corresponding layer tree.

The layer tree is a data structure which can store multiple open kinematic chains. The nodes in a layer tree are the *layers*. Such a layer can be

- a solid object which is delimited by two parallel planes and the lateral area joining the two planes (a polyhedron);
- a solid object which is symmetrical with respect to an axis (for time efficiency and computational simplicity, such a solid is approximated by a polyhedron);
- empty.

Each layer has a coordinate system attached to it and two associated transformation matrices:

- A transformation matrix C coming from the kinematic description. This is normally a transformation matrix derived from the Denavit-Hartenberg parameters of the robot which is modeled.
- The specification of the solid parts of the link is done as follows:
  - If the solid object is delimited by parallel planes then one plane is the x, y-plane. The polygon in the x, y-plane is specified plus the height of the solid object.
  - If the solid object is symmetrical with respect to an axis then this axis is the x-axis. The intersection of the object with the x, y-plane is a polygon. This polygon is specified. It is allowed that only a half of it is specified because it will always be symmetrical with respect to the x-axis.

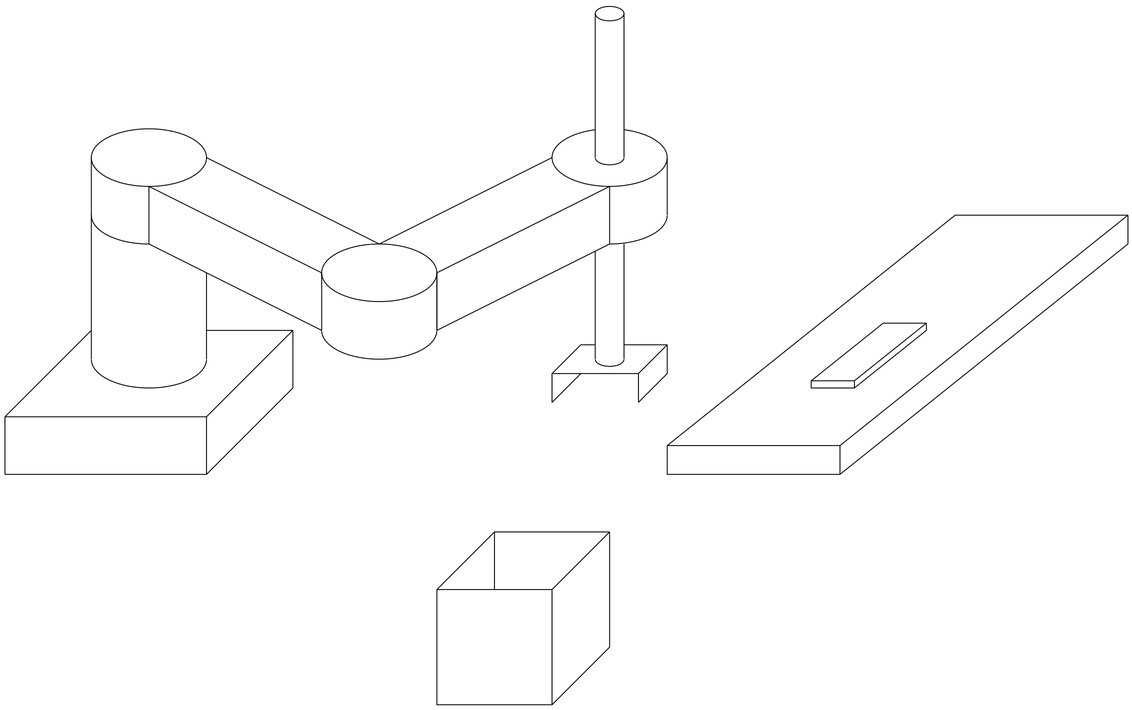


Figure 3.1: A robotic manipulator of the Scara type.

> Following the explanations of Section 1.3.1 each link of a robotic manipulator has a fixed coordinate system, the *link coordinate system*, (which is at a proper place determined by the convention following the Denavit-Hartenberg parameter approach, see Section 1.3.1). Unfortunately, it is rarely the case that the solid of the link is in a position that it can be specified as above with respect to the link coordinate system. The dilemma is solved by specifying a constant matrix S which moves and rotates the solid into the correct position with respect to the link coordinate system.

The overall transformation of a layer is, therefore, $C \cdot S$, i.e. the layer's children are transformed by this amount.

When building the layer tree for a real robot, it is a good idea to model the kinematic chain in the Denavit-Hartenberg sense separate from the "flesh" of the robot. We achieve this as follows: For each link we introduce a dummy layer which is empty, the *link reference layer*. Its coordinate system will be the link coordinate system of the link. The link is moved by changing the kinematic matrix C of the link reference layer. Since only the kinematic frames contribute to the kinematic chain, the layer's S-matrix must be the identity matrix. On the other hand, the layers modeling the solid parts of the robot are children of the link reference layer. Their kinematic matrix C must be the identity matrix and the S-matrix is such that the solid is positioned correctly with respect to the link coordinate system (i.e. the link reference layer).

The layer tree of the scene in Fig. 3.1 is shown in Fig. 3.2. The layers in dashed frames are the link reference frames. The others model the solid parts of the robot. (From this example we also see that we need more than one solid for each link.)

The layer tree is the input model of SAVE. The layer tree representation of a robot is specified in a file which the SAVE system reads in. For collision detection and for the manufacturing simulation a hybrid internal model is implemented. The solid models of

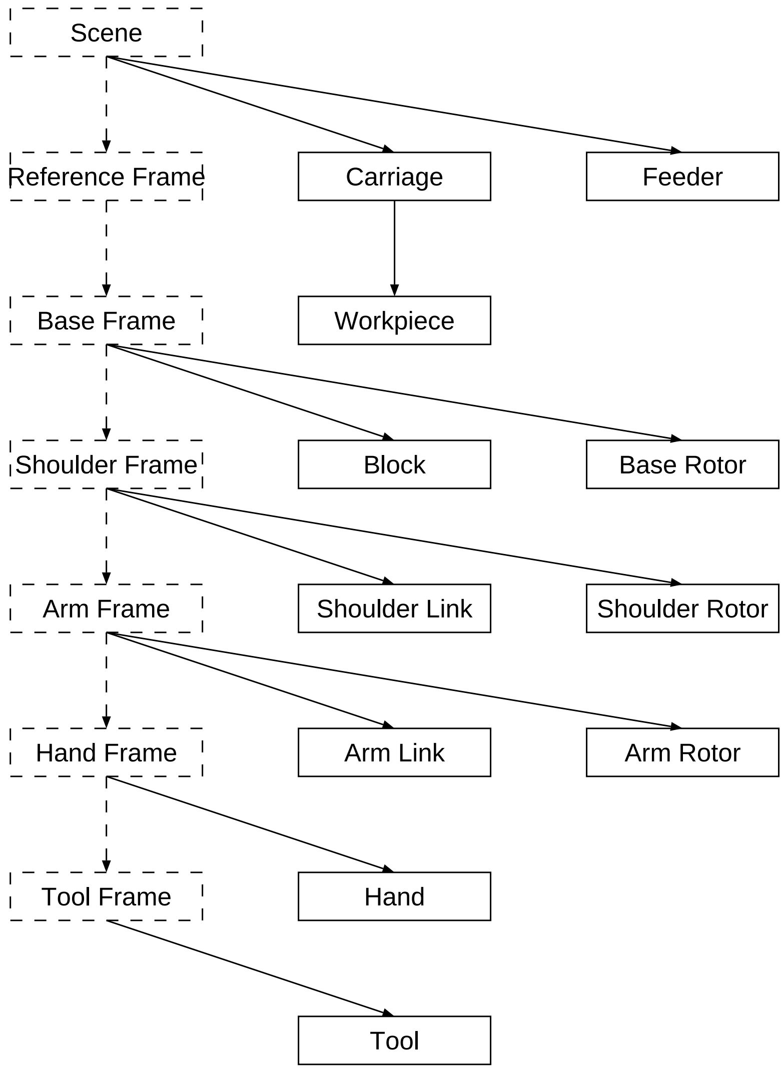


Figure 3.2: The layer tree of the Scara robot of Fig. 3.1.

the layer tree are converted to CSG (Constructive Solid Geometry) [Män88] and Dexel (Depth Element, analogous to pixel, Picture Element) models [MH91].

For the output, i.e. for displaying the manipulator on a screen, SAVE utilizes *PHIGS* (*Programmers Hierarchical Interactive Graphics System*) [ISO87]. The layer tree is converted to a PHIGS structure hierarchy. PHIGS is an ISO standard with a defined interface to the programming language C.

Chapter 4

A Concept for a New Version of NETUSE

A concept for a new version of the previously described ANN simulator NETUSE is developed and proposed. Our concept concentrates on a new design of a command interpreter in this chapter. The next chapter will present a flexible ANN simulator using the new tool. Both a user interface and a programming interface are specified for the command interpreter. Section 4.2 provides a reference of the command language for the user, that is, the person who is using an application which incorporates the language interpreter. Section 4.3 is the reference for application programmers, who extend the language with new commands, functions, and object classes.

One main improvement over the existing command interpreter NETUSE will be that we not only specify a new command interpreter, but a full *language*, in which programs can be written. Within the frame of a language, commands are only a special case of statements which are issued interactively. The language has flexible control structures like if-statements, different kinds of loops (for-, while-, do-while-loop), functions and procedures which take parameters. Variables of different types (three different sizes of integers, two different floating point formats, strings) can be declared and used in complex expressions.

The second major improvement will be that the language is *object oriented* allowing inheritance of methods, which strongly encourages re-using of existing code. However, classes and methods can only be *used* in the interpreter. New classes cannot be constructed within the frame of the language, but only by an application program which registers new commands and classes to the interpreter.

We mention here that our partners released an improved version of NETUSE, called NEUROtools, which incorporated some ideas that came up during early discussions. In our approach, we do not resort to this work.

4.1 Analysis of Experiences

The new concept shall incorporate new features which were deemed useful and eliminate flaws which showed up in earlier releases of NETUSE (see the following sub-sections).

It turned out that an *interpretative command interface* is extremely useful. It shall be kept and its capabilities shall be reviewed and expanded. In particular, only with an interpreter it is possible to write command files, which is often The command interface in

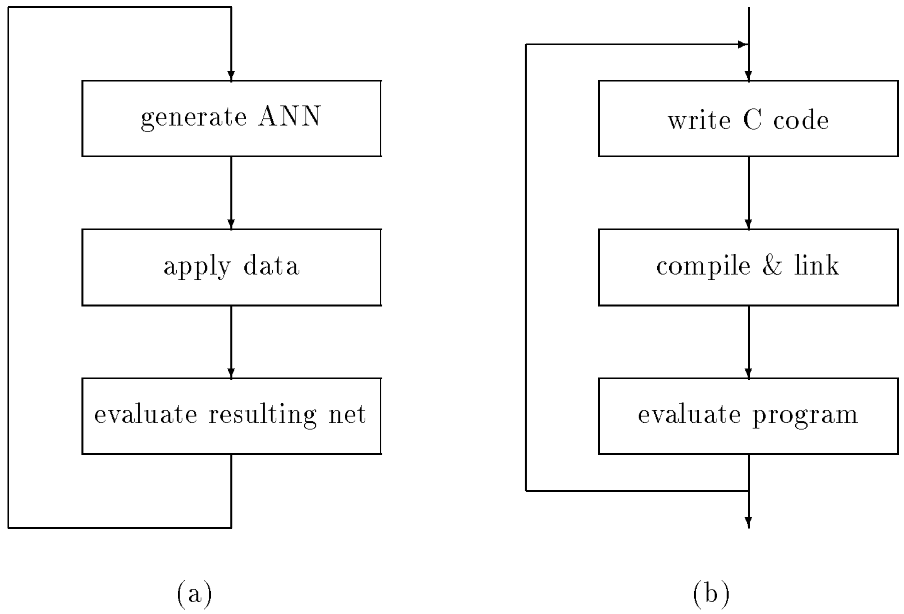


Figure 4.1: (a) shows the life cycle of an ANN during the development of an ANN solution. Often the steps have sub-cycles as in (b) because parts of the simulator have to be coded in C.

earlier versions turned out to be too inflexible. Fig. 4.1(a) shows the cycle an ANN undergoes during the development of a net solving a particular problem. For many problems, however, this cycle does not remain that trivial. During each step it is often necessary to introduce new views, distributions, or external commands, i.e. C source code has to be written, compiled, and debugged. Therefore, each step has sub-cycles as shown in Fig. 4.1(b).

This situation is due to the inflexible commands that were implemented so far: It is difficult for the developer to find an existing command that implements the behavior he needs because they were specialized for a single task or type of network.

The external commands written so far are weakly documented. Hence it is difficult to copy an existing function and then to adjust it. It often happens that a function does not work any more after some small changes because of the strong dependencies among the functions provided by the kernel.

4.1.1 Advantages of Earlier Versions

Extensibility. The developer is not restricted to the set of commands provided by the command interpreter. Rather, he can implement new commands and tell the interpreter about them, so that they can be invoked.

Modularity. The command interpreter is a module by itself. It was designed to be reused in different applications, which can provide different commands. (Although the module is not completely independent in earlier versions, independence is of course desirable.)

4.1.2 Disadvantages of Earlier Versions

No functions. To modularize a command-"program" it is necessary to split it into several command files. The result is that many little files clutter the working directory. It is difficult to keep the overview on the functionality of the modules.

No nested loops. The command interpreter does not provide a construct to get nested loops directly. To achieve nested loops it is also necessary to write a command file which contains the inner loop (which is the body of the outer loop). The outer loop must have an invocation of that command file in its body, which is dramatically time consuming.

Line-orientedness. One line of input is a basic unit for the command interpreter: Loop commands repeats exactly one line; the if-command can conditionally execute one line. (For this very reason the if-command does not have an else-option.)

Lack of flexible types. Variables can only take integers. For many applications, however, floating point types are desirable, and strings would be helpful. Structured types could often help to increase the readability of command files (which in turn makes such programs easier to maintain).

4.2 The Command Language

The basic task of the command interpreter so far was simply to provide a dispatch mechanism to the external commands. The interpreter itself had only little functionality like expression evaluation, simple looping, and macros. In order to avoid the time-consuming write-C-code/compile-link/debug sub-cycles during the development it is desirable to have much more functionality in the command interpreter. The intention is that it should be necessary to write C code in only few situations.

To achieve this goal the command interpreter will be extended to a *command language* featuring types, variables, expressions, classes with methods and instantiation, built-in commands, external commands, procedures, and a "shell escape". The command language will be named *NCI6t* because it evolves from the NETUSE Command Interpreter and it would be version 6 of NETUSE.

This section specifies the interface of the command language to the user, i.e. the syntax and semantics of the language, and the interface to the programmer who wants to implement external commands, classes and methods.

In contrast to other interpreted languages, NCI6t is a *strictly typed* language and can, therefore, easily be compiled to efficient machine code. Interpreted languages like dBase III [PH87] commonly involve dynamic (implicit) declarations. Such features are not desired when a language is being compiled because the types of variables and their sizes and addresses in memory are not known during compile time so that many errors occur only at runtime and cannot be detected by the compiler (cf. Clipper [Hoc89]).

4.2.1 Syntax and Semantics

The command language, as seen by the operator (user), is specified in this section. For the specification of syntax the *Extended Backus Naur Formalism* [Wir85] is used. Therein, syntactic entities are written in normal text font, "`teletype font`" enclosed in quotes is used for verbatim characters and words. Parts enclosed in brackets, [ ], are optional, parts enclosed in curly braces, { }, can be repeated zero or more times, and | denotes choice. Parentheses, ( ), are used for grouping.

The Alphabet of the Language

Every program is a *word* consisting of *letters* from an *alphabet*. The alphabet of NCI6t consists of identifiers, numbers, strings, special-character-tokens, and reserved words. These parts will be defined in the following paragraphs. The letters of the alphabet are also named *lexical tokens* or simply *tokens*.

Token classes. Identifiers, numbers, and strings belong to token classes. They have the following sub-syntax:

identifier = (letter | “_”) { letter | digit | “_” }.
number = digit { digit } [“.” { digit }
[(“E” | “e”) [“+” | “-”] digit { digit }]].
string = “"” { *any printable character except* “"” *and* “\” | “\n” | “\t” |
“\f” | “\"” | “\r” | “\\” } “"”.
letter = “a” | “b” | ... | “z” | “A” | “B” | ... | “Z”.
digit = “0” | ... | “9”.

In a string “\n” is the new-line character, “\t” the tab character, “\f” the form-feed character; “\"” inserts a double quote character, “\r” a carriage-return character, and “\\” a backslash.

Numbers which contain a dot “.” are floating point constants. They can optionally be followed by an exponent specification “E” or “e” which denotes “... times 10 to the power of ...”. Numbers without a dot are integer values.

Tokens. Some sequences of one, two, or three non-digits and non-letters are tokens. These are the special-character-tokens:

```
<   <=   ==   !=   >=   >    (    )     [    ]     {    }    ?    ??
#   +    -    *    /    %    &    &&    |    ||    ^    .    ,    ;
=   +=   -=   *=   /=   %=   &=   &&=   |=   ||=   ^=   @    !    ~
```

Reserved words fall within the syntax of identifiers. However, they have special meanings and cannot be used like other identifiers. These are the reserved words:

```
and    break  bye    byte    const   delete   do     double  else
exit   float  for    help    if      include  int16  int32   new
not    or     quit   return  shell   string   void   while   xor
```

They may not be used as variable names or function names. Applications which define classes cannot use them as class names, member names, or method names.

White space. Blanks, tab characters, new-line characters, carriage-return characters, and form-feed characters have no meaning whatsoever and are ignored, except that they separate tokens.

Comments. Comments can be given in one of two ways:

- The character sequences "`/*`" and "`*/`" delimit comments. These delimiters nest, i.e. for each opening "`/*`" a closing "`*/`" must occur in the program code.
- Everything on a line following the character sequence "`//`" (including these characters) is treated as comment.

A comment is treated as a blank character, i.e. it is ignored, except that it separates tokens.

Scope of Names and Life Time of Variables

Every identifier that is used in an NCI6t program must be declared. The context within which the declaration occurs determines the scope of the identifier and how long the entity denoted by the identifier exists. NCI6t employs the *lexical-scope rule* with a "most closely nested" stipulation added, cf. [ASU86, Chapter 7]. This rule is also used in languages like Pascal or C.

Space for a variable is allocated at the time its declaration is encountered. The memory space occupied by a variable is kept until the end of the variable's scope is reached. The scope is the body of the function (from the opening brace "`{`" to the closing brace "`}`"), if the variable is declared inside the body of a function, or global if the variable is declared outside of functions (and, therefore, ends when the command interpreter terminates).

When inside a function a name is used which is neither a formal parameter nor a local variable, it is assumed that the name denotes a global entity (variable or function). In this case, the actual semantics of the name is determined at a later time. The current meaning of the name is used when the function is executed.[1]

Types and Variables

A *data type* represents a set of values which a variable of this type can hold. NCI6t supports integers, floating point numbers, strings, arrays, and classes.

Integers NCI6t supports three integer types:

- "`byte`" values are in the range 0 to 255;
- "`int16`" values are in the range -2^{15} to $2^{15} - 1$;
- "`int32`" values are in the range -2^{31} to $2^{31} - 1$.

Floating point numbers can have 2 formats: "`float`" values occupy 4 bytes of memory, "`double`" values occupy 8 bytes of memory. Floating point numbers are encoded as specified in the IEEE standard 754 [IEE87].

Strings of characters with a maximum length of 255 characters.

Instances of classes which are provided by external modules.

[1] Strictly speaking, every (global) name should have got a unique meaning (by declaring it), because redeclaration should not be allowed in a strictly typed language. However, during interactive work it is inevitable to redefine a name. For example, if the type of a variable has been declared wrong, it should be possible to correct this by redeclaration. Therefore, in the actual implementation of the command interpreter, the meaning of a name may indeed change.

Boolean values are represented by integers: 0 means *false*, any other value means *true*.

A collection of values of the same basic data type can be arranged in an *array*. Individual array elements can be accessed by *indexing*. For this purpose, array elements are numbered from 0 up to the number of elements minus one.

Expressions can handle *function values*. For this purpose, a *function type* exists. Variables and formal parameters can be declared to take functions as their values.

Furthermore, applications using NCI6t can provide *classes* of objects. They represent collections of values of different types plus methods that operate on these values. A variable can be declared to hold an *instance* of a class. There do not exist useful standard classes. The only built-in class, Core, provides an interface between NCI6t and applications which subclass this class. It has no other functionality.

The types "`byte`", "`int16`", "`int32`", "`float`", "`double`", and "`string`" are termed *simple types*. Classes and arrays are termed *compound types*.

Declarations. Variables are declared using the following constructs:

variable-decl	=	["`const`"] type var-decl-list.
type	=	"`byte`" | "`int16`" | "`int32`" | "`float`" | "`double`" | "`string`" | "`void`" | identifier.
var-decl-list	=	var-decl { "`,`" var-decl }.
var-decl	=	identifier [array-type | function-type].
array-type	=	"`[`" [expression] "`]`".

The expression in an array declaration must evaluate to a positive integer. It specifies the number of elements for which the array provides space. This expression *must* be specified in a variable declaration. It can be omitted in a formal parameter declaration. In this case the formal parameter array has as many elements as the actual parameter has.

If the type-name is an identifier it must name a class supplied by the application.

Type conversion. The integer and floating point types can be ordered as follows:

byte < int16 < int32 < float < double.

If during the evaluation of an expression the two arguments of a binary operator are of different types, the lower type is converted to the higher, so that the two arguments have the same type. The result of the operation is then of that type. If one of the types is not numeric, the operation is invalid and generates an error.

If a value is assigned to a variable by an assignment statement or a formal parameter by passing it as an actual parameter, and the types of the value and the variable or formal parameter are different, the value is converted to the type of the variable or formal parameter. This conversion obeys the following rules:

- Integer → floating point. This is straight forward.
- Integer → string. The integer value is converted to its string representation using radix 10.
- Floating point → integer. If the floating point value is outside the range of the integers, the behavior is implementation dependent. Otherwise, the value is truncated at the decimal point (rounded towards zero).

- Floating point → string. The floating point value is converted to its string representation with an implementation dependent accuracy.
- String → integer. The string is parsed as the syntax of “number” specifies, up to, but not including the decimal point “`.`”. Leading white space is ignored. Parsing terminates as soon as an unrecognized character is encountered.
- String → floating point. The string is parsed as the syntax of “number” specifies. Leading white space is ignored. Parsing terminates as soon as an unrecognized character is encountered.
- All other conversions are invalid and generate an error.

Expressions

NCI6t understands expressions involving constants, variables, functions, and operators for arithmetic operations, logical operations, comparisons and string operations.

The syntax of expressions is as follows:

expression = and-expression { or-operator and-expression }.
or-operator = “`or`” | “`||`” | “`|`” | “`xor`” | “`^`”.
and-expression = comparison { and-operator comparison }.
and-operator = “`and`” | “`&&`” | “`&`”.
comparison = arithmetic { comp-operator arithmetic }.
comp-operator = “`<`” | “`<=`” | “`==`” | “`!=`” | “`>=`” | “`>`”.
arithmetic = [add-operator] term { add-operator term }.
add-operator = “`+`” | “`-`”.
term = factor { mul-operator factor }.
mul-operator = “`*`” | “`/`” | “`%`”.
factor = constant | “`(`” expression “`)`” | (“`not`” | “`!`” | “`~`”) expression | “`new`” identifier parameters | var-or-func-call | “`#`” identifier.
constant = number | string.
var-or-func-call = variable parameters.
variable = identifier { “`.`” identifier | “`[`” expression “`]`” }
parameters = [“`(`” [expression { “`,`” expression }] “`)`”].

This syntax implies the precedence of the operators.

Arithmetic Operators. The operators “`+`”, “`-`”, “`*`”, and “`/`” are addition, subtraction, multiplication, and division, respectively. They operate on numeric values. The operator “`%`” is the remainder of the division of the arguments. It operates only on integral values. The remainder has the same sign as the dividend (the left operator).

Logical Operators. The operators “`and`” (synonymous to “`&&`”), “`or`” (synonymous to “`||`”), and “`not`” (synonymous to “`!`”) are boolean operators. They operate on integers. “`and`” and “`or`” are “lazy-evaluated”, that is:

x `and` y is equivalent to if x then y else 0
x `or` y is equivalent to if x then 1 else y

The operators “`&`”, “`|`”, “`xor`” (synonymous to “`^`”), and “`~`” are bitwise logical operators. They operate on integers, whose binary representation is used for theses operations. The result is the bitwise and, bitwise inclusive or, bitwise exclusive or, and bitwise negation, respectively.

Comparisons. The operators “`<`”, “`<=`”, “`==`”, “`!=`”, “`>=`”, and “`>`” are comparisons. The result is 1 if and only if the left operand is less than, less than or equal to, equal to, not equal to, greater than or equal to, and greater than the right operand, respectively. They operate on integers, floating point numbers, and strings. Strings are compared lexicographically using machine dependent character codes. Country specific characters may or may not be ignored.

String Operator. “`+`” is the only string operator. It concatenates its arguments. If the result is longer then 255 characters, it is truncated at the right to 255 characters.

Element count operator. The operator “`#`” determines the number of elements of its argument, which must be the name of an array variable or formal array parameter. This operator is useful if the number of elements in a formal array parameter is not specified explicitly, but implicitly by the corresponding actual parameter.

Operands. The operands of expressions can be constants, variables, elements of arrays, members of class instances, and values returned from function calls and method calls.

Applications may define *symbolic constants*. These are named by an identifier. The type of a symbolic constant is “`int32`”. Each symbolic constant is assigned a unique value which is constant while NCI6t is running, but which is not guaranteed to be identical from one run of NCI6t to the next. Symbolic constants are intended to be stored in “`int32`” variables, to be passed as parameters to commands, functions, and methods, and to be tested for equality. Other operations (addition, other comparisons, printing the value, ...) are allowed but have no meaning, since the value of a constant may change from one run of NCI6t to the other.

Square brackets “`[`” and “`]`” denote array indexing. The expression is evaluated and is converted to an integer if it is not already an integer. The variable which is indexed must be an array. If the resulting index is outside the bounds of the array (less than zero or greater than or equal to the number of elements of the array) an error is generated.

The dot “`.`” denotes component selection. It is used to select members and methods of the instance which is denoted by the variable's part before the dot.

Round parentheses “`(`” and “`)`” which immediately follow a variable specification denote the invocation of a function or method. The function being invoked must not be declared to return a “`void`” value. The *actual parameters* of the function or method are evaluated in order and substituted for the formal parameters. The actual parameter is assigned to the formal parameter; the actual parameter value is converted to the formal parameter's type if the type does not match. There must be exactly as many actual parameters as formal parameters.

NCI6t provides a set of standard functions. Depending on the application which incorporates NCI6t, additional functions may be provided. These are the standard functions:

- Trigonometric functions “`sin`”, “`cos`”, “`tan`”, “`asin`”, “`acos`”, “`atan`”. They take one floating point parameter in the usual ranges. If the parameter lies outside the allowed ranges an error is generated. "`atan2`" takes two floating point parameters. An error is generated if both parameters are zero. All trigonometric functions return a floating point value.

- The exponential function "`exp`" takes one floating point parameter x and returns e^x. The power function "`pow`" takes two floating point parameters x and y and returns x^y int the usual definition of the exponential function. If both parameters are zero an error is generated.

- The random function "`rand`", which takes no parameters, generates a different integral pseudo-random number each time it is called in the range 0 to $2^{31} - 1$. It can be seeded with the "`seed`" command. The period of the random numbers is implementation dependent, but at least $2^{31} - 1$.

- Shell escape function "`shell`". The argument to this function must be a string. It is passed to the `popen()` function call. The result of this function is the standard output of the command (a string with at most 255 characters). The function generates an error if the exit code of the invoked program is non-zero.

Parameters of functions and commands. Values of the simple types are always passed *by value* to functions and commands. Arrays of any type and class instances (compound types) are always passed *by reference*.

Functions

The syntax for a function declaration is partly shown in Section 4.2.1, variable declarations.

function-type	=	"`(`" [formal-param-list] "`)`" [function-body].
formal-param-list	=	formal-parameter { "`,`" formal-parameter }.
formal-parameter	=	["`const`"] type var-decl.
function-body	=	"`{`" var-or-stat-list "`}`".
var-or-stat-list	=	var-or-stat { ["`;`"] var-or-stat }.
var-or-stat	=	variable-decl | statement.

The body (i.e. the statement list) of a function may contain "`return`" statements, which terminate the execution of the function and may return a value to the caller. If during the execution of the function the end of the body is reached, the function implicitly executes a "`return`" without an expression.

If "`const`" is specified in front of a formal parameter this parameter cannot be changed in the body of the function by an assignment. Specifying simple type parameters as "`const`" has no real use because such parameters are passed by value, and changing the formal parameter does not change the actual parameter. For compound types, however, the "`const`" keyword specifies, that the elements of the formal parameter cannot be changed by an assignment, thus the actual parameter, which is passed by reference, cannot get changed, either. However, if the formal "`const`" parameter is an instance of a class, the members of the object may indirectly be changed if a method is invoked on this object and the method changes members. A "`const`" formal parameter may neither be used as the left-hand-side of an assignment nor as actual parameter of a function whose corresponding formal parameter lacks the "`const`" keyword.

In a variable declaration inside a function new functions cannot be declared (i.e. function declarations cannot be nested). Similarly, in the formal parameter list the specification of a function body is not allowed.

If a formal parameter is an array, the expression specifying the number of elements can be omitted. In this case, the number of elements of the actual parameter will be used as the number of elements in the formal parameter. The "`#`" operator can be applied to the formal parameter name to query the number of elements in the actual parameter.

Names which were declared outside the function (*global* names) are known inside the function. The formal parameters and all names that were declared inside the function are *local* names. These may "shadow" global names, thereby making the global object inaccessible.

Statements

An NCI6t program is a sequence of statements:

statement-list	=	statement { ["`;`"] statement }.
statement	=	[assignment | if-statement | while-statement | do-while-statement | for-statement | "`return`" [expression] | "`break`" | command].

The syntax provides for an empty statement. If an empty statement is encountered, nothing is executed and control is passed on to the next statement. Also note, that the semicolon between statements is optional.

Assignments. The assignment has the syntax

assignment	=	variable assign-op expression.
assign-op	=	"`=`" | "`+=`" | "`-=`" | "`*=`" | "`/=`" | "`%=`" | "`&=`" | "`&&=`" | "`|=`" | "`||=`" | "`^=`".

The normal assignment operator is "`=`". The special assignment

`v` *op*`= e`

is equivalent to the normal assignment

`v = v` *op* `(e)`

where *op* is one of the arithmetic, logical, or string operators, `v` is a variable and `e` is an arbitrary expression. The special assignments may perform faster than the explicit normal assignment. Note, that the expression is evaluated before *op* is applied so that `x *= y+1` is equivalent to `x = x * (y+1)` and not `x = x * y+1`.

If the type of the expression does not match the type of the variable, the expression's value is converted to match the type of the variable. The current value of the variable is replaced by the value of the expression after its evaluation (and possible conversion). If the variable is a formal parameter of a function, but not a component of a formal parameter array or class instance, the value of the formal parameter is changed and the corresponding actual parameter is left unchanged (call by value). If the variable is a component of a formal parameter array or class instance, the corresponding actual parameter is also changed (call by reference).

Conditional statement. The conditional statement is the if-statement:

if-statement = "`if`" "`(`" expression "`)`" "`{`" statement-list "`}`"
{ "`else`" "`if`" "`(`" expression "`)`" "`{`" statement-list "`}`" }
["`else`" "`{`" statement-list "`}`"].

The expressions are evaluated in sequence until one results in a non-zero value. Then the corresponding statement list is executed. If all expressions evaluate to zero the statement list after the "`else`" keyword is executed, if present. Otherwise, nothing is executed.

Loop statements. The while-, do-while-, and for-statements execute their corresponding statement lists repeatedly:

while-statement = "`while`" "`(`" expression "`)`" "`{`" statement-list "`}`".
do-while-statement = "`do`" "`{`" statement-list "`}`" "`while`" "`(`" expression "`)`".
for-statement = "`for`" "`(`" [assignment] "`;`" expression "`;`" [assignment] "`)`"
"`{`" statement-list "`}`".

The while-statement repeats the statement list zero or more times as long as the expression evaluates to a non-zero value. The do-while statement executes the statement list at least once as long as the expression evaluates to a non-zero value. The for-statement

```
for (a1; e; a2)
    { statement-list }
```

is equivalent to the while statement

```
a1;
while (e)
    { statement-list;
      a2 }
```

where a_1 and a_2 are possibly empty.

Return statements. When a return statement is executed, the current function is terminated and control is passed back to the calling function immediately after the point where the current function was called. If the function was called during the evaluation of an expression the value of the return statement's expression is the result of the function. A return statement must not occur outside of any function.

Break statements. The break statement passes control to the statement immediately following the end of the innermost loop statement enclosing the "`break`". A break statement must not occur outside of any loop (not even inside a function body which is later potentially executed inside a loop).

Commands

The main purpose of NCI6t is to execute commands interactively. From the user's point of view there exists a minimum set of commands that NCI6t understands. Depending on the application into which NCI6t is incorporated additional commands may be provided. The syntax of commands is:

command	=	("`include`" | "`@`") expression |
		("`shell`" | "`!`") [expression] |
		"`seed`" expression |
		("`?`" | "`??`") [expression-list] |
		"`delete`" expression |
		"`exit`" | "`quit`" | "`bye`" |
		variable [expression-list].
expression-list	=	expression { expression }.

The set of standard commands is described in detail below.

Including files. The "`include`" command (synonymous to "`@`") executes commands in a file. The expression must evaluate to a string, which is used as the file name from which to read commands. The effect is as if the "`include`" command were replaced by the commands in the file. In particular, no new scope for names begins.

Shell escape. If the "`shell`" command (synonymous to "`!`"[2]) is used without an argument, an operating system shell is opened. The environment variable `SHELL` specifies the shell to use. If this variable is not set, some implementation dependent default will be used. The command is finished when the invoked shell terminates. If an argument is provided to the "`shell`" command, the argument must result in a string. Then this string is passed on to the `system()` function call.

Seeding the random number generator. The "`seed`" command sets the random number generator's seed value to the value of the expression. This command is used to reproduce a sequence of pseudo-random numbers.

Printing values. The output commands "`?`" and "`??`" print their arguments on the terminal. The expressions must evaluate to simple types. Non-string values are converted to strings. Nothing is printed between the values. The command

`?` $e_1 \ldots e_n$

is equivalent to

`??` $e_1 \ldots e_n$ `"\n"`

Deleting class instances. The "`delete`" command is used to destroy class instances. The expression must evaluate to a class instance.

Exiting. The commands "`exit`", "`quit`", and "`bye`" are synonyms. Their behavior is, in general, application dependent. The intention is that all necessary cleanup is performed and the application is signalled to terminate.

[2]Since in a command file the new-line character does not terminate a command, the "`shell`" keyword must be preferred over "`!`" because the latter can result in in ambiguous syntax: The command sequence

```
? x
! "ls"
```

will result in an error like "integer value required". The error is triggered by the string `"ls"`, which is treated as the argument to the Boolean operator "`!`".

Other commands. Functions declared to have a return type "`void`" can be invoked as commands. Additional commands may be provided by the application. These can be *normal commands* or methods of classes. A normal command is invoked by specifying its name followed by arguments, if required. Methods are invoked by specifying the instance onto which the method shall operate followed by "`.`", the method name, and additional arguments required by the method.

Classes and Instances

NCI6t supports structured types called *classes*. These classes can be *instantiated*. It depends on the application which uses NCI6t if there exist classes. It is not possible to define classes or to subclass existing classes with the language NCI6t.

A variable that is explicitly declared as a class instance is not instantiated immediately. Rather, it must be assigned an instance created from the appropriate class with the "`new`" construct. Instances created in this way must be destroyed with the "`delete`" command. If new instance is assigned to a variable which holds already an instance, that instance is *not* destroyed.

A class may have *methods* associated with it which operate on an instance of the class plus arguments which are passed to the methods when they are invoked.

A method can be used in one of two forms: as command or as function. A method is accessed by denoting the instance of the class followed by a dot "`.`" and the name of the method, followed by command arguments or actual parameters, respectively. Invoking a method on an instance is also called "sending a message to the instance".

Methods cannot be handled by expressions and can, therefore, not be assigned to variables or passed as parameters like functions.

There exists only one standard class, Core, which is not very useful for the user. It only implements the basic functionality necessary for the class programmer to develop customized classes.

Constructor and destructor. Every class has at least two special methods, the *constructor* and the *destructor*. The constructor is automatically invoked when a new instance of the class is created with "`new`". The parameters specified with "`new`" are passed to the constructor. The destructor is invoked when the instance is destroyed (with "`delete`"). The destructor does not receive additional arguments.

A Session

An NCI6t session, finally, is a sequence of variable and function declarations and statements (which include commands):

$$\text{session} \quad = \quad \text{var-or-stat-list.}$$

The session represents the global level of an NCI6t program.

4.3 The Programming Interface

The command language specified in the previous section provides a basic framework for an application with an interpretative command interface. The functionality of of the language is kept at a minimum, but it is possible that an application using the language extends the language by new commands, functions, and object classes by implementing them in C. For this purpose, an interface to the command language at the C language level is defined in this section. For example, many of the built-in functions and some of the commands are actually implemented using this interface.

We specify two requests which must be satisfied by the programming interface of the command language:

1. The command language must be a module by its own which does not depend on functions or data of applications which use the module. The command language is a library which can be used like a "toolkit".

2. It is desirable that the new commands and functions not only have an interface to the command language, but also a C language interface. This makes it possible that application functions can call others directly and that there is no overhead in the calling sequence.

Goal 1 is achieved as follows: The module maintains a symbol table into which the application registers its extensions. The functions associated with command names are passed by pointers. Goal 2 cannot be reached in a simple and clear way. So the decision was made to put a little overhead into the C language interface of application functions for the sake of clearness.

4.3.1 The Structure of an NCI6t Application

From the application's point of view the flow of control is fairly different in an NCI6t application from a traditional command line oriented application. Traditional applications perform in the following way:

1. The operator is asked which operation shall be performed.

2. Depending on the chosen operation further information may be necessary. In several prompts the operator is asked to provide the information.

3. Finally the operation is performed and control returns to step 1.

In an NCI6t application the control is different:

1. The application first performs its own initializations (if necessary). Additionally, it also initializes the NCI6t library (toolkit).

2. Then the commands implemented by the application are registered, so that the command interpreter knows about them.

3. The application now calls the function NciInterpret and transfers the overall control to the NCI6t module. This function implements the command interpreter and dispatches the commands, functions, and objects implemented by the application. The function returns when the user executes the "`exit`" command.

   The additional information necessary for an operation (*command*) is provided as *parameters* to the operation, but the command itself does not suspend execution to wait for further information.

4. All necessary clean-up is performed.

This scheme is very similar to how an application with a graphical user interface would look like. Indeed, it would well be possible to replace the command line interface by a graphical interface, and the application would essentially be programmed in the same way.

4.3.2 Naming Conventions

To make it easy for the programmer to avoid typographical errors in C code the following conventions for naming objects shall apply:

- All functions, global variables, types, and constants exported by the NCI6t library begin with the letters "Nci". The base name of the function, variable, type, or constant is usually not abbreviated unless it would become too clumsy. If the name consists of several words the individual words are capitalized and are pasted together without any intervening characters. Underscores "_" are not used.

- Members of structures are written in lower case letters. The words in names consisting of several words are separated by underscores "_".

4.3.3 An Example of a Class Implementation

An object implementation consists of two parts: a class header file, which can be used by other class implementations which want to extend the functionality of the class by sub-classing; and the actual implementation, the C source code.

The interface of a class consists of two parts: One part can be used by C code to use the class directly: by calling methods or by accessing fields of an instance. The other part is used by the main program to register the class to the command line interpreter.

We show an implementation of the FloatData class as specified in Section 5.1.2. It is a subclass of Data, which we assume to be already implemented.

The class header file. This file specifies the interface of the class to the application. The first part of the header file includes the superclass's header file unless the class is directly derived from the core class.

```
/* FloatData.h - class header */

#ifndef _FloatData_h
#define _FloatData_h

#include <Data.h>
```

Here a preprocessor macro is defined to prevent errors if the header file is included multiple times. We also include the header file of the superclass, which is `Data.h` in our case.

```
typedef struct {
    NciMethod process_floats;
} FloatDataClassPart;

#define InheritProcessFloats _NciInherit
```

This part defines the *class part structure*. It holds the elements specific for the new class, which is a pointer to the ProcessFloats method for the FloatData class. A constant is defined, InheritProcessFloats, that can be used in subclasses to indicate that the subclass does not want to implement a new ProcessFloats method but wants to inherit the superclass's method. (_NciInherit is a procedure which, when invoked, issues an error message saying that inheritance resolution was not performed correctly.)

```
typedef struct {
    CoreClassPart core_class;
    DataClassPart data_class;
    FloatDataClassPart float_data_class;
} FloatDataClassRec, *FloatDataClass;

extern FloatDataClassRec floatDataClassRec;
extern Class floatDataClass;
```

The next part defines the whole *class structure* for the new class. The class structure must contain all superclass's parts (in order) as well as its own class part. The variable floatDataClassRec will be used for subclassing, the variable floatDataClass will be used when a new instance of the class is allocated to designate the class.

```
typedef struct {
} FloatDataPart;
```

The following part defines the *instance part structure*, which holds the values specific for an instance of this class. These are the new members of the instance which are filled by the methods and which may be accessed directly. The FloatData class has no new data in the instance, so we leave the instance part structure empty.

```
typedef struct {
    CorePart core;
    DataPart data;
    FloatDataPart float_data;
} FloatDataRec, *FloatDataObject;
```

This defines the *instance structure* of the new class. It must hold all superclasses' instance part structures (in order) as well as the class's own structure.

```
typedef struct {
    NciValue src;
    NciValue first_src, num_records, first_dst;
    NciValue process_proc;
} ProcessFloatsParam;

typedef struct {
    NciValue src;
    NciValue dst;
} ProcessFloatsProcParam;
```

We must define parameter structures for all formal parameter lists which will be necessary. In our case, these are the parameters of the new method and the formal parameter list of the function which is passed to this method. The members of formal parameter structures must always be of type NciValue, because they will be used to type-cast a pointer to an *array* of NciValues to the formal parameter *structure*. By this the parameters can be accessed by name (using the structure's member names) rather than by indexing in the array.

So far, we have specified the first part of the class's interface: the part which will be used for subclassing and to access the class from C code. The second part of the interface, which will be used by the main program to register the class to the command interpreter, is extremely simple, because it is only a single variable:

```
extern NciClassSE floatDataClassSE;

#endif    /* _FloatData_h */
```

The variable will hold a description of the class for use by the command language interpreter. The variable can be used by the application to register the class to the NCI6t library and by subclasses. Note, that the name of the class as it will be known to the command interpreter is not contained in any of these records. It will only be fixed when the class is registered to the interpreter. As a result, the command interpreter knows about the functionality (i.e. the methods and members) of the class *and* all its superclasses, but it only knows a name for the class, but *not* for the superclasses (unless they are registered separately).

The implementation. Here a possible (simplified) implementation of the FloatData class is shown.

```
/* FloatData.c - class implementation */

#include <stdio.h>
#include <stdlib.h>     /* malloc, free */
#include <macros.h>     /* min */
#include <NCI6t.h>
#include "FloatData.h"

static void ClassInitialize(Class c);
static void ProcessFloats(Object obj, NciValuePtr ret_val, NciValuePtr par);

FloatDataClassRec floatDataClassRec = {
  {     /* core_class */
    (Class) &coreClassRec,     /* super_class */
    sizeof(FloatDataRec),     /* instance_size */
    ClassInitialize,     /* class_initialize */
    FALSE,     /* initialized */
    NciInheritConstructor,     /* constructor */
    NciInheritDestructor     /* destructor */
  },
  {     /* data_class */
    sizeof(NciFloat),     /* unit_size */
    DEFAULT_MAX_REC_NO,     /* max_rec_no */
    InheritLoad,     /* load */
    InheritSave,     /* save */
```

```
    InheritCopy,          /* copy */
    InheritSetMaxRecNo,       /* set_max_rec_no */
    InheritGetPointer,        /* get_pointer */
  },
  {       /* float_data_class */
    ProcessFloats         /* process_floats */
  }
};

Class floatDataClass = (Class) &floatDataClassRec;
```

First of all, the class structure is defined and initialized. We do not provide our own implementations of the superclasses' methods, so we inherit them by specifying the Inherit...-constants. Note, that we must specify an implementation of the new ProcessFloats method because it is newly introduced in the FloatData class. Subclasses of this class can specify InheritProcessFloats for this value of the class structure to indicate that they do not want to implement their own versions of this method but rather use the superclass's method.

```
static void ClassInitialize(Class c)
{
     FloatDataClass fdc = (FloatDataClass) c;

    /* Task 1: One-time initializations of FloatData class */
    if (!floatDataClassRec.core_class.initialized) {
        /* initialize class part of superclass */
        floatDataClassRec.core_class.super_class→
                        core_class.class_initialize(floatDataClass);
        floatDataClassRec.core_class.initialized = TRUE;
    }

    if (c == NULL)
        return;

    /* Task 2: class part initializations of class in parameter c */
    floatDataClassRec.core_class.super_class→
                        core_class.class_initialize(c);

    /* inherit methods of FloatDataClassPart */
    if (fdc→float_data_class.process_floats == InheritProcessFloats) {
        fdc→float_data_class.process_floats = ((FloatDataClass)
                fdc→core_class.super_class)→float_data_class.process_floats;
    }
}
```

The class initialization function has two tasks:

1. It must perform all one-time initializations required for the FloatData class (but not its subclasses). Most importantly, this means that the class record gets initialized. Further initializations will usually concern static variables used by methods implemented in the class.

2. It must initialize the class part structure float_data_class of the class passed as parameter. This most commonly includes the resolution of method inheritance.

```
static void ProcessFloats(Object obj, NciValuePtr ret_val, NciValuePtr par)
{
#define FIELD(fdo, field) ((FloatDataObject) fdo)→data.field
#define METHOD(fdo, meth) ((FloatDataClass)((Object)
fdo)→core.class)→data_class.meth
    int n;      /* number of records to process */
    ProcessFloatsParam *p = (ProcessFloatsParam *) par;
    ProcessFloatsProcParam proc_p;

    if (!NciIsSubclass(p→src.object_val, floatDataClass)) {
        NciError("source must be a FloatData object");
    }

    n = min(p→num_records.int32_val,
                FIELD(p→src.object_val, num_records) - p→first_src.int32_val);
    if (n ≤ 0)
        return;     /* nothing to do */

    p→first_src.int32_val += n, p→first_dst.int32_val += n;     /* begin at back */
    for (; n > 0; n--) {
        proc_p.src.array_val.data = METHOD(p→src.object_val, get_pointer)
                (p→src.object_val, --p→first_src.int32_val, 0, DataDontExtend,
                &proc_p.src.array_val.num_elements);
        proc_p.dst.array_val.data = METHOD(obj, get_pointer)
                (obj, --p→first_dst.int32_val, 0, DataExtend,
                &proc_p.dst.array_val.num_elements);
        p→process_proc.func_val.proc(p→process_proc.func_val.data, NULL,
                (NciPointer) &proc_p);
    }
#undef FIELD
#undef METHOD
}
```

The methods which are new to this class and methods which override methods of the superclass are then implemented. We only have to code the ProcessFloats method.

The calling interface to other application code is now complete. The remaining part of the implementation specifies the interface to the command interpreter.

```
static NciParamSE process_floats_proc_param_se[] = {
    { "src", { NciFloatT | NciArrayT | NciConstT, 0, NULL },
                                NciOffset(ProcessFloatsProcParam, src) },
    { "dst", { NciFloatT | NciArrayT, 0, NULL },
                                NciOffset(ProcessFloatsProcParam, src) }
};

static NciFormalParSE process_floats_proc_formal_se =
```

```
    { { NciVoidT, 0, NULL }, sizeof(ProcessFloatsProcParam),
                process_floats_proc_param_se,
NciNumber(process_floats_proc_param_se) };

static NciParamSE process_floats_param_se[] = {
    { "src", { NciObjectT, 0, &floatDataClassSE, NULL },
                                NciOffset(ProcessFloatsParam, src) },
    { "fs", { NciInt32T, 0, NULL, NULL },
                                NciOffset(ProcessFloatsParam, first_src) },
    { "nr", { NciInt32T, 0, NULL, NULL },
                                NciOffset(ProcessFloatsParam, num_records) },
    { "fd", { NciInt32T, 0, NULL, NULL },
                                NciOffset(ProcessFloatsParam, first_dst) },
    { "proc", { NciFunctionT, 0, NULL, &process_floats_proc_formal_se },
                                NciOffset(ProcessFloatsParam, process_proc) }
};

static NciFormalParSE process_floats_formal_se =
    { { NciVoidT, 0, NULL }, sizeof(ProcessFloatsParam),
                process_floats_param_se, NciNumber(process_floats_param_se) };

static NciMethodSE float_data_meth_se[] = {
    { "ProcessFloats", NciOffset(FloatDataClassRec, float_data_class.process_floats),
                &process_floats_formal_se }
};

NciClassSE floatDataClassSE = {
    &dataClassSE,
    (Class) &floatDataClassRec,
    NULL,
    float_data_meth_se, NciNumber(float_data_meth_se),
    NULL, 0
};
```

Fig. 4.2 shows the interconnection among these structures.

4.3.4 An Example of a Main Program

The following shows an example of a main program using the class implemented in the previous section.

```
/* main program */

#include <stdlib.h>
#include <NCI6t.h>
#include "FloatData.h"

void main(int argc, char *argv[])
{
```

```
    NciInitialize();

    NciRegisterClass("FloatData", &floatDataClassSE);

    NciInterpret();

    exit(0);
}
```

4.3.5 Initializing NCI6t

NCI6t must be initialized before an application can use it. For initializing NCI6t use the function

void NciInitialize(**void**);

This function sets up the symbol table and registers some built-in functions. It also determines if commands are read from a file or from a terminal.

4.3.6 The Symbol Table

During a run of NCI6t numerous names are declared and used. In order to keep track of the semantics of an identifier a *symbol table* is maintained. It stores of which kind individual identifiers are and which types they have. The kind and type determines how an identifier can be used.

The symbol table is altered in different situations:

- During the initialization with NciInitialize some built-in names are registered.
- When the application initializes, further names can be registered by the application's extensions with NciRegisterConstant, NciRegisterVariable, and NciRegisterClass.
- When the command interpreter encounters a function declaration or variable declaration, the names declared also enter the symbol table.

Many entries of names in the symbol table have a type associated. This can be one of the following:

```
typedef void NciVoid;
typedef unsigned char NciByte;
typedef short NciInt16;
typedef long NciInt32;
typedef float NciFloat;
typedef double NciDouble;
typedef char *NciString;
typedef struct {
        NciFunctionProc proc;
        NciPointer data;
} NciFunction;
typedef struct {
        NciPointer data;
        int num_elements;
} NciArray;
```

(This assumes that short variables are 16-bit signed and long variables are 32-bit signed, which is true for most of the Unix systems currently in use.) The type of the variable is described by

```
typedef struct {
    NciTypeId type_id;      /* base type */
    int num_elems;      /* number of elements if the base type is an array */
    /* (we cannot use a union below since they cannot be initialized) */
    struct _NciClassSE *class;      /* class if NciObjectT */
    struct _NciFormalParSE *formal_par;      /* parameters if NciFunctionT */
} NciType;
```

where NciTypeId is a value of the enumeration

```
typedef enum {
    NciVoidT, NciByteT, NciInt16T, NciInt32T, NciFloatT, NciDoubleT,
    NciStringT, NciFunctionT, NciObjectT
} NciTypeId;
```

The values

```
#define NciArrayT 0x80
#define NciConstT 0x40
```

may be or'ed into a value of type NciTypeId. NciArrayT indicates that the value is an array. NciConstT indicates that the value cannot be changed by an assignment.

The value of a variable is stored in a memory location of type NciValue:

```
typedef union _NciValue {
    NciByte byte_val;
    NciInt16 int16_val;
    NciInt32 int32_val;
    NciFloat float_val;
    NciDouble double_val;
    NciString string_val;
    NciFunction func_val;
    Object object_val;
    NciArray array_val;
} NciValue;
```

Registering Names to the Symbol Table

Before registering identifiers to the symbol table some data structures must be set up to contain information about the identifier.

When the application tries to register a symbol table entry having a name which has already been registered (by NciInitialize or the application itself) with another symbol table entry, the application is aborted.

Registering variables. To register a variable use NciRegisterVariable:

```
void NciRegisterVariable(char *name, NciType *type, NciValue *value);
```

Registering a variable only makes the name known to the symbol table at the outermost level of the command interpreter. If the value of the variable requires additional memory to be allocated, this must be done by the caller:

- If the value is an NciString, additional memory must be allocated (usually by calling NciNewString), which can hold the string. The member string_val must be adjusted to point to this memory. It is not allowed to initialize the variable with a string constant unless the type of the variable is or'ed with NciConstT.
- If the value is an Object, the memory for this object is not allocated immediately. The NciNewObject function must be used to allocates the memory and to initialize the instance.
- If the value is an array, memory for the array must be allocated. This must not be an array of NciValues, but an array of memory cells of the indicated type.

The type of the variable must not be NciVoid. The type may be or'ed with NciConstT to indicate that the value of the variable may not be changed by an assignment. In this way, "symbolic constants" can be declared.

Registering functions. A function in NCI6t is treated as a constant value of a special type, namely of a function type having a specific set of formal parameters and specific return type. Therefore, a function is registered with NciRegisterConstant.

Before a function can be registered, the formal parameter table must be set up, which is NciFormalParSE:

```
typedef struct _NciFormalParSE {
        NciType return_type;     /* the type of the return value */
        int param_size;
    /* the size of the memory which holds the parameters */
        NciParamSE *params;      /* the array of parameter descriptions */
        int num_params;      /* the number of parameters */
} NciFormalParSE;
```

return_type specifies the type of the returned value. It may be NciVoid, indicating that no value will be returned. The return type must not be an array nor an Object. params specifies the formal parameters of the function. It points to an array of parameter symbol table entries NciParamSE:

```
typedef struct {
        char *name;      /* the name of the parameter */
        NciType type;      /* the type of the parameter */
        int offset;      /* offset of the value in the parameter structure */
} NciParamSE;
```

num_params specifies the number of the formal parameters. The order in which the parameters must be specified when the function is called is given by the order in the array of parameter specifications.

When the function is executed, the actual parameters will be provided in an array of NciValues to the function. Since the command interpreter cannot know the semantics of each parameter (which is an element of the array) it puts the actual parameters in it in the order as they are specified in the actual parameter list (provided that the parameter types match the formal parameter specifications). The offsets specified in the NciParamSE must be taken from a structure consisting only of NciValues which will be used by the implementation of the function to access the actual parameters by overlaying it onto the array provided by the command interpreter. See the example on page 49 for how offsets are computed. During the registration of the function the following checks are performed:

- The offsets in the parameter specification must be increasing.
- The difference between two successive offsets must be equal to the size of an NciValue.
- The total space required by the parameters in the specification must be equal to the size of the parameter structure, which is specified in NciFormalParSE.param_size.

If any one of these checks fails, the application program will abort immediately. After these checks, the offsets are no longer needed. The checks are done to avoid errors which may arise if the parameter structure is extended, rearranged, or otherwise changed, but the programmer forgets to also change the parameter description array. The correctness of the types cannot be checked in any way, so this matter is on the programmer's behalf only.

As with constants in general, the implementation of a function must provide a correctly initialized NciValue for registration. The data part can be initialized with an arbitrary value. It will simply be passed along to the function (as its first parameter) when it is invoked. The proc part of the value must be initialized with the function pointer of type NciFunctionProc:

typedef int (*NciFunctionProc)(NciPointer data, NciValuePtr ret_val, NciValuePtr par);

The data parameter of the function will contain what is provided in the data part of the NciValue.func_val.data. The return value can be put into the NciValue whose pointer is passed in ret_val. This pointer shall not be used under any circumstances if the function is declared to return NciVoid. The par parameter will point to the array of actual parameters. As explained above, the parameter values should not be accessed by index, but the pointer should be cast to an appropriate structure pointer, so that the parameters can be accessed "by name" inside the implementation of the function.

Writing Classes (Subclassing)

Classes, instantiation of classes, inheritance, and overloading are extremely powerful and flexible concepts in software engineering. They encourage reusing already written modules while it is still possible to enhance the functionality of existing program code without rewriting all of it.

NCI6t uses the following mechanism for implementing classes:

- Each class has a description record which holds the methods of the class as function pointers.
- Every instance knows the class to which it belongs. This is not trivial. Consider the following case: The FloatData class is a subclass of Data class, which defines a Copy method. We assume that the FloatData class overloads it with its own method. A

variable *d* of type Data class can also take an instance of FloatData class (because the latter is a subclass of the former). We now want to invoke the Copy method of *d*. If we do not know the actual class of the instance referenced by *d*, the only thing we know would be, that it is a Data class. So we could only invoke Data.Copy, which is not correct because the actual class is FloatData class, which has its own Copy method.

- NCI6t defines only one class, CoreClass, which provides only administrative functionality for inheritance and method overloading.
- When a class is subclassed the subclass's implementation must only care for the new parts which it introduces.

Class parts and class records. The subclass must define a record which can hold the methods and per-class data which is *new* for this class. This record is called *class part*. The subclass then collects the class parts of all its superclasses and its own class part in the *class record*. The class part of CoreClass is:

```
typedef struct {
        Class super_class;
        int instance_size;
        NciClassProc class_initialize;
        int initialized;
        NciMethod constructor;
        NciObjectProc destructor;
} CoreClassPart;
```

The class record of CoreClass is simply:

```
typedef struct _CoreClassRec {
        CoreClassPart core_class;
} CoreClassRec, *CoreClass;
```

The new class must allocate a single static *class record variable* and initialize it with the relevant data. The field core_class.initialized must always be initialized with FALSE. All method fields must get assigned a method. The special "method" Inherit*method-name* can be used to indicate that the method *method-name* is inherited from the superclass.

In addition to the class record variable a *class variable* must be defined which is initialized with the address of the class record variable. The class variable has the generic type Class, which is a pointer to CoreClassRec, the class record of the core class. (Therefore, when the class variable is initialized, the pointer to the new class's class record variable must be type-cast to Class; see the example implementation above.) Whenever a class is required as a parameter to a function in application code, the value of the class variable can be used in a simple way—no type casts and address operations (&) are necessary any more. NciNewObject is an example for a commonly used function which takes a class as parameter.

It is not allowed to leave the class part away if a new class does not need new per-class data. In this case an empty class part structure must be defined.[3] As a result in the

[3]Some C compilers do not allow empty structures, so a dummy element can be provided to keep the compiler quiet.

class record the class parts of *all* superclasses are listed. (By this, one always sees the superclasses of a class at the first glance.)

Instance parts and instance records. The data structure of the instance is partitioned analogously to the class data. A subclass must define an *instance part* which can hold the *new* data used by the class. Then all instance parts are collected into the *instance record*. The basic class CoreClass has only one field in its instance part:

```
typedef struct {
        Class class;
} CorePart;
```

Analogous to its class record, the instance record is simple:

```
typedef struct _CoreRec {
        CorePart core;
} CoreRec;
```

When a new instance of a class is created, the class member of the CorePart will be initialized to point to the class record of that class. This is how the instance knows its class. All method calls will use this pointer to look up the class record, where the method (a function pointer) is registered.

Registering classes. Before a class can be registered, a set of symbol table entries must be set up: The parameters for the constructor, the methods and their parameters, and the accessible class members must be provided. Fig. 4.2 shows how the data structures are interconnected.

The description of a class symbol table entry is:

```
typedef struct _NciClassSE {
        struct _NciClassSE *super_class_se;
        Class class;
        NciFormalParSE *constr_params;
        NciMethodSE *methods;
        int num_methods;
        NciParamSE *members;
        int num_members;
} NciClassSE;
```

super_class_se points to the symbol table entry of the superclass. class points to the class record variable of the corresponding class. If the class defines a new constructor, constr_params points to a formal parameter description analogously to a method's formal parameter description. These will be the parameters to the "`new`" operator.

The methods field points to an array of NciMethodSE's, whose number of elements is given in num_methods: These describe the newly introduced methods and overloaded existing methods. The structure of an NciMethodSE is:

```
typedef struct {
        char *name;        /* the name of the method */
        unsigned int offset;      /* the offset of the method in the class record */
        NciFormalParSE *param;      /* the formal parameters of the method */
} NciMethodSE;
```

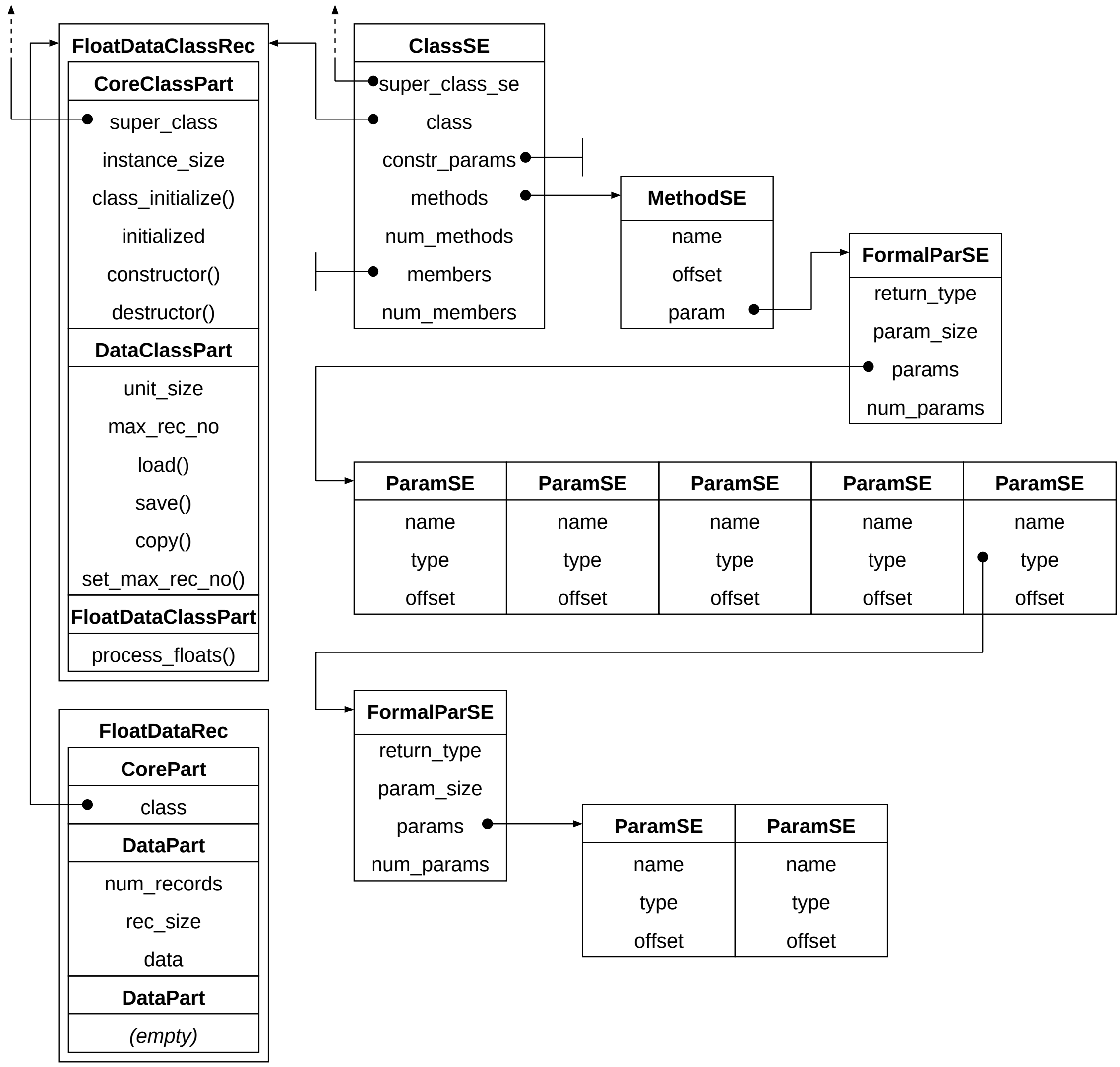


Figure 4.2: The data structures describing an object, its accessible members, and methods. As an example the FloatData class as described in Section 5.1.2 is used.

The field name specifies the name of the method, the field param a description of its formal parameters. The offset is the offset of the methods function pointer in the class record (which is pointed to by NciClassSE.class).

To register a class use NciRegisterClass:

void NciRegisterClass(**char** *name, NciClassSE *class);

where the parameter name eventually specifies the name of the class.

4.3.7 Methods

A class may define *methods*. These are functions which operate on an instance of that class, thereby possibly changing the contents of the members of the instance.

Since C does not have a class concept, and hence no notion of methods, we implement methods as functions. When a method is invoked it gets a pointer to the instance as a parameter, so it can access the contents of the instance.

In NCI6t a class can have *external* and *internal* methods. External methods can be invoked by the command interpreter. Since the command interpreter must pass parameters to the method, it must have a unique interface. Therefore, the type of external methods must be

typedef void (*NciMethod)(Object obj, NciValuePtr ret_val, NciValuePtr par);

The object on which the method shall operate is passed in the parameter obj. The parameters are passed to the method as an array of NciValues, which is pointed to by par, which is the same as with functions. The method may return a value in the NciValue pointed to by ret_val.

Internal methods are only accessed by C code. Therefore, the parameter list of an internal method can be arbitrary, except that the object on which it shall operate is passed as the first argument.

For each new method which is introduced by the class a constant Inherit*method-name* must be introduced (in the class header file). The constant is the function _NciInherit type-cast to the new method's function prototype (if necessary):

void _NciInherit(Object obj, NciValuePtr ret_val, NciValuePtr par);

It is used in the initialization of the class record to designate that the method shall be inherited from the super class. The class initialization method can recognize this and must copy the superclass's method to the subclass's method field. If the developer of the new subclass codes its class initialization method erroneously so that no inheritance takes place, _NciInherit will be called instead of the correct method. _NciInherit is a function which simply issues an error and aborts the application since it is, of course, a fatal condition that the method was not inherited correctly.

Note, that if Inherit*method-name* is used to initialize a method pointer in the class record, it does *not* mean

> "Inherit the method from the class which defined Inherit*method-name*."

It rather means

> "Inherit the method from the superclass,"

which may already be a different method. This further means, that it is not necessary for a subclass which exchanges a method to define a new Inherit*method-name* macro.

For each external method a formal parameter description must be set up, which is exactly the same as for functions. Also a parameter structure should be used in the implementation of the method, so that it is possible to access the parameters by name and not by an index in the parameter array.

It is good style to introduce dummy-methods in classes which are intended to be subclassed. By this, a unique name for a method is introduced. The user does not need to remember the different names of the methods in different subclasses. Some applications of this technique are shown in the next chapter. For example, the Layer class introduces a Recall-method (which implements an ANN's recall phase) as a dummy. The dummy method will then be replaced in the subclasses by their appropriate methods. In this way, Recall can be performed in an RCM-Layer with the same name as in the backpropagation Layer, "Recall". This technique, however, must be avoided if the interface, i.e. the parameters, to the method must be changed in subclasses. It is confusing and a sure source of errors if methods in different classes have different interfaces, but the same name!

Methods of the Core Class

The core class of NCI6t provides three methods: the class initialize method, the constructor, and the destructor. The core class defines the two constants

```
#define NciInheritConstructor _NciInherit
#define NciInheritDestructor ((NciObjectProc) _NciInherit)
```

which can be used to inherit the constructor and the destructor from the superclass. The class initialization method cannot be inherited; every class must have its own class initialization method.

Class initialization. The class initialization method is a function of type NciClassProc:

typedef void (*NciClassProc)(Class class);

Its major task is to initialize the class part of the class record which is pointed to by the parameter class. This task is usually only the resolution of inheritance, which means that the function pointers for methods which contain the constant _NciInherit must be filled in with the superclass's method.

The structure of class initialize methods is normally very similar. After type-casting the argument to the class's class record structure, the method first of all checks if the class for which it is implemented is already initialized. If it is not, it calls the superclass's class initialize method to fill in all superclasses' class parts of its own class record. This is also a good point to perform one-time initialization tasks. Then it checks if the argument is NULL. If it is, the method is done; its task was to initialize the own class record. Otherwise, it must now initialize the argument class record. For this, the superclass's class initialization method is called with the argument passed along. This will initialize all class parts of the argument belonging to the superclasses. Then the own class part of the argument must be processed, which means that all method pointers are check if they are _NciInherit and if so, to replace that value with the argument's superclass's method pointer.

To explicitly initialize a class use

void NciClassInitialize(Class class);

This function first checks if the class record pointed to by the parameter is already initialized. If it is, the function immediately returns. Otherwise, it calls the class's class initialization method with an argument of NULL.

Creating an instance. A new instance of a class is created with

Object NciNewObject(Class class, NciValuePtr params);

The parameter class specifies the class of the new object. This value can be taken from the class's class variable (which will then not require a type cast); it must be a pointer to the class record of the desired class in any case. In params an array of parameters is supplied analogously to function parameters. This function is usually only called by NCI6t, in which case the parameters will be taken from the list supplied with the "`new`" operator. Only under rare circumstances it will be necessary that the application calls this function.

NciNewObject performs the following:

1. It is first checked, if the class is already initialized. If it is not, NciClassInitialize is called with class as argument.

2. Memory is allocated to hold the instance. The size of the instance is taken from the CoreClassPart of the class record.

3. The class field of the CorePart in the newly allocated instance is initialized to the parameter value class. (Now the instance knows the class to which it belongs.)

4. The constructor of the class is invoked. For this, an NciValue is allocated and the int32_val field is initialized to zero. Then a pointer to the new instance, a pointer to this value and the parameter array received from the caller are passed along to the constructor.

5. If upon return from the constructor the int32_val field is non-zero the new instance is deallocated and a null pointer is returned to the caller (the initialization of the instance failed). Otherwise, the pointer to the new instance is returned to the caller.

The constructor method is of the type NciMethod and is an external method. The obj parameter specifies the new object instance. As the first thing, the constructor must call the superclass's constructor, possibly after setting up necessary parameters. Then it must check ret_val.int32_val if the superclass's constructor succeeded. If this field is non-zero, the superclass's constructor failed and the constructor must return immediately. If it succeeded, the constructor continues. Its task is to initialize the fields in the instance part structure of the instance, using the parameters specified in par. It may also modify fields belonging to any superclass's instance part structure. If some initialization cannot be performed, the constructor can set ret_val.int32_val non-zero to indicate failure. In this case it must call the superclass's destructor and return.

Destroying an instance. To delete an instance of a class use

void NciDeleteObject(Object obj);

The function calls the destructor of the class and then deallocates the memory occupied by the instance.

The destructor method is of the type

typedef void (∗NciObjectProc)(Object obj);

and is an internal method. The obj parameter specifies the object being destroyed. The destructor's task is to release all dynamically allocated resources that were allocated for the instance during its lifetime. It must call the superclass's destructor immediately before it returns.

4.3.8 Miscellaneous Functions

Invoking the Command Interpreter

The function

void NciInterpret(**void**);

invokes the command interpreter. It reads commands from the standard input channel until it encounters an "`exit`", "`quit`", or "`bye`" command or the end of the input file.

Testing the Class of an Instance

To test whether an object instance is of a specific class or a subclass thereof, use the function

int NciIsSubclass(Object obj, Class class);

The function returns zero if the instance specified by obj is not a subclass of the class specified by class. Otherwise, it returns a non-zero value.

Working with Strings

In NCI6t, strings are simple types, that is, they are treated exactly like integers and floating point numbers, as far as parameter passing and assignment is concerned. Since in C strings are implemented as arrays of characters, which cannot be treated like numbers, we must emulate the simple-type-behavior of strings in NCI6t.

The critical points in this treatment concern assignments and parameter passing.

- When a number is assigned to a variable, the value is actually copied into the location of the variable. When dealing with strings, we must, therefore, also copy the string into the variable's location.
- Simple types are passed by value to functions. For this reason, we must allocate a new copy of the string and pass the copy to the function.

For working with strings, the NCI6t toolkit pre-allocates memory which can hold strings with the maximum length of 255 characters. To seize an instance of a string from this pool, use

char ∗NciNewString(**void**);

The function returns a pointer to a location which can hold up to 255 characters and the terminating null-byte. To release a string and give it back to the pool, use

void NciDelString(**char** $*$str);

The function caller must allocate a copy of a string which is passed as an actual parameter. It is the function's responsibility to deallocate it. If a function returns a string, it must allocate a string, fill it with the result and return it to the caller.

When a new variable is allocated by a declaration, a string from the pool will also be allocated. Hence, when a string value is assigned to a variable, the value must be copied to the variable's string memory.

Many optimizations are possible to reduce the number of allocations and deallocations of strings. For example, if a function has a string argument and returns a string, it may be possible that the function does not deallocate the string it received in the parameter, but reuses it for the result string. Similarly, when a string s is assigned to a variable v it is not necessary to copy the string. Strictly following the instruction above, the string must be copied from the source memory s.string_val to the destination memory pointed to by v.string_val, then the source memory is deallocated. In a more relaxed way, the destination memory is deallocated and the pointer s.string_val is assigned to v.string_val.

In order to avoid fatal situations, it is absolutely necessary that there never exist two pointers (in C-global memory) which point to the same string memory.

Error Messages

To issue an error use the function

void NciError(**char** $*$text, ...);

The function prints an implementation dependent string which identifies the message as an error, followed by the text specified in the argument text, followed by a new-line to the standard error channel. The argument text can contain formatting instructions as they are used in the standard C function "printf". Additional arguments may follow the text argument in this case. The text argument should not end with a new-line character because it is appended automatically.

Chapter 5

Generating and Operating Artificial Neural Networks

In this chapter we will concentrate on how NCI6t can be used for establishing an ANN simulation environment. As there exist many paradigms of artificial neural networks with different behavior in both the neurons and the synapses, generating a network will be a major concern. The user portion of this part will be covered in Section 5.1.1. In Section 5.1.2 we will describe NCI6t classes and functions which allow to work with a network. In Section 5.2 we will develop the implementations of those classes and functions.

5.1 User Interface of ANN Classes

One major restriction in the implementation we provide will be that there can be only one network in the memory at any time. The reason for this can be found in the dual view of networks: When we operate a network we want to access only parts of the network, for example, we want the hidden layer of a network to perform a training step. On the other hand, when we want to store the state of a network, we want to do it for the whole network, and not only for a part of it. It is difficult to implement this duality for a single network, and it would not pay to implement it for multiple networks, since in practice working with one network at a time is sufficient.

5.1.1 Generating an ANN

The commands, objects, and functions we will develop imply a certain structure in the ANN to be created.

- The ANN is layered. Connections may exist between neurons of different layers as well as among the neurons of a single layer. The neurons of a layer will always be identical in their structure and functionality. This does not only help to logically structure the network (by keeping identical neurons together), but also has the side effect, that these neurons can be treated efficiently in computation. Of course, different layers can have neurons of different type.

- The neurons of a layer are considered to be arranged in a 3-dimensional grid. (1-dimensional and 2-dimensional layers are simply treated as special cases of the 3-dimensional grid.) Such geometric arrangements of neurons are often extremely natural; e.g. a picture is clearly 2-dimensional. It is not only easier for the user to create a new network (or understand an existing one) if the neurons can be viewed as arranged on a grid; most importantly, it is the basis for an effective network generation process using view, distributions, and projections (as explained below).

Numerous different *neuron models* exist, for example, the backpropagation neuron, the Hopfield neuron [Roj93]. Generally, each neural network paradigm uses its own neuron model, which, from the programmer's point of view, is characterized by the data structure needed to parametrize the neuron and the functionality the neuron must provide to be able to work in a neural network.

For the representation of a neuron model by NCI6t we will separate the specification of parameters from the specification of functionality. Parameters are part of the *neuron type*, functionality is associated with the *layer*. This separation is only for practical purposes and efficiency: We can change the behavior of a layer by simply assigning it a different neuron type. For this reason, the implementations of the layer and the neuron type are always closely related. For every neuron model a new layer class and a new neuron type class will be necessary.

For the specification of the network topology we want to preserve the flexibility that NETUSE provided. The main point in the method (invented by Dr. Hans Geiger) is that the local structure of a network can be expressed (cf. the definition of a *view* given below). This is a powerful tool which allows to build complicated structures like network cascades (cf. the feature extracting network Fig. 5.3), but we also can implement standard network topologies (like backpropagation).

Of basic importance in the following definitions are the notions of presynaptic and postsynaptic neurons. Let us consider a single neuron N. The set of *presynaptic* neurons with respect to N are those neurons from which N receives input. The set of *postsynaptic* neurons with respect to N are those neurons which receive input from N.

The following data structures (view, distribution, projection) are used to describe the connections between neurons in a network. They are typical constituents arising in Geiger's network paradigm. Since it is not the intention of our work to study this paradigm with all its special features, for a detailed description of these notions we refer to [Was87]. For the convenience of the reader we briefly summarize in an informal style the definition of these notions.

- Given the coordinates of a neuron, the *view* defines a subset of neurons of a layer with respect to this *origin*. Optionally, some constant parameters may further describe the subset. Each view has a layer associated with it from which the neurons are selected. Different views can have the same layer associated with them. The view is used to describe the network locally.

- The *distribution* connects a given neuron with the set of neurons selected by a given view. These are then presynaptic neurons. The origin of the view is found by a suitable transformation of the given neuron into the presynaptic layer (see Eq. (5.1)).

- The *projection* applies a distribution to all neurons specified by a (postsynaptic) view. The origin of this view is defined by specifying a fixed postsynaptic neuron.

An exact definition of the above notions can alternatively be extracted from the pseudo code definitions given in Section 5.2.2

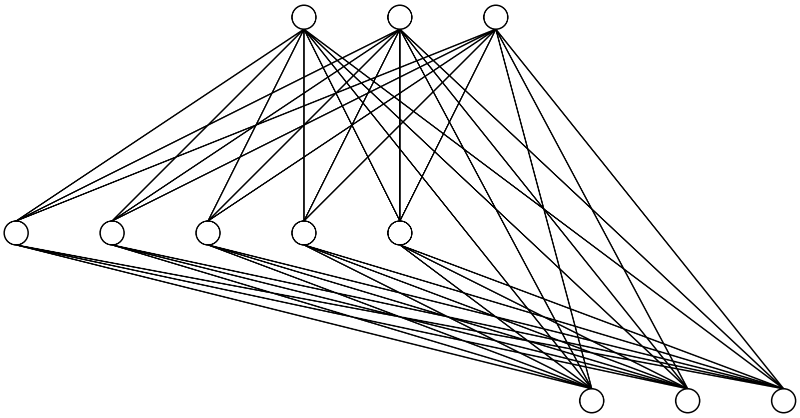


Figure 5.1: A three layered net with direct connections between input and output layer.

As an example, we present the NCI6t command script which generates the network topology of Fig. 5.1, assuming that the necessary classes are implemented. The network consists of three layers which are fully connected between the input and hidden layer, the hidden and output layer, as well as directly between the input and output layer. (We use backpropagation neurons, but this is not of relevance for the topology.)

```
/* neuron types                     learning parameter, momentum */
NeuronType nt_input  = new BPNeuronType(0.0, 0.0);
NeuronType nt_hidden = new BPNeuronType(0.3, 0.5);
NeuronType nt_output = new BPNeuronType(0.3, 0.5);

/* layers                neuron type x  y  z */
Layer input  = new BPLayer(nt_input,   3, 1, 1);
Layer hidden = new BPLayer(nt_hidden,  5, 1, 1);
Layer output = new BPLayer(nt_output,  3, 1, 1);

/* a view function which selects all neurons in a layer */
int view_select_all(View view, int x, int y, int z) {
    return 1; /* every neuron is selected */
}

/* generate views */
View view_input  = new View(input,  view_select_all, 0.0);
View view_hidden = new View(hidden, view_select_all, 0.0);
View view_output = new View(output, view_select_all, 0.0);

/* generate distributions */
Distribution dist_input  = new Distribution(view_input);
Distribution dist_hidden = new Distribution(view_hidden);

/* generate projections and net                        origin for view */
Projection proj_in_hid = new Projection(dist_input, view_hidden, 0, 0, 0);
Projection proj_hid_out= new Projection(dist_hidden,view_output, 0, 0, 0);
Projection proj_in_out = new Projection(dist_input, view_output, 0, 0, 0);
```

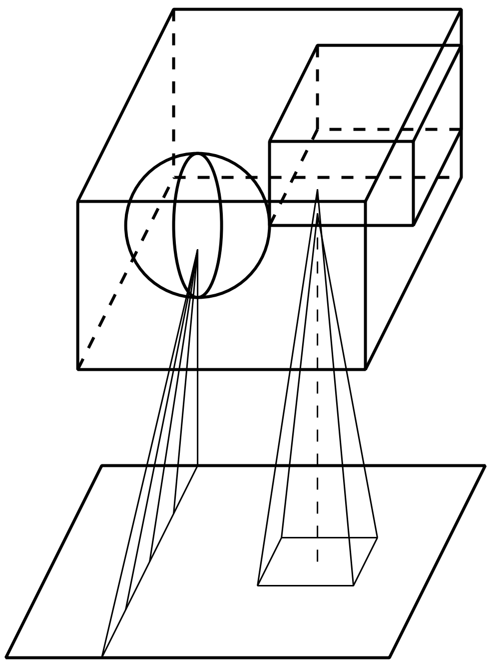


Figure 5.2: A neural network with non-standard presynaptic and postsynaptic sets of neurons.

After this code is executed, the network is built.

Next we consider the more complicated (but "pathological") example in Fig. 5.2 to show some more sophisticated views. The presynaptic layer is 2-dimensional, the postsynaptic layer is 3-dimensional. Each postsynaptic neuron in the spherical subset receives input from the neurons of the presynaptic layer whose x-coordinate corresponds to the x-coordinate of the postsynaptic neuron. Each neuron in the cubic subset receives input from the subset of neurons which are arranged in a square whose center corresponds to the x- and y-coordinates of the postsynaptic neuron.

```
/* neuron types */
NeuronType nt_input  = new BPNeuronType(0.0, 0.0);
NeuronType nt_output = new BPNeuronType(0.2, 0.3);

/* layers */
Layer input  = new BPLayer(nt_input,  20, 20,  1);
Layer output = new BPLayer(nt_output, 15, 15, 15);

/* view functions */
/* This function selects a plane parallel to y-z-plane */
int view_select_x(View view, int x, int y, int z) {
    return (x == view.x_origin);
}

/* This function selects cube with given size. Origin specifies center. */
int view_cube(View view, int x, int y, int z) {
    int h = view.param / 2.0;  /* param specifies length of side */
```

```
    return (
        view.x_origin - h <= x && x <= view.x_origin + h &&
        view.y_origin - h <= y && y <= view.y_origin + h &&
        view.z_origin - h <= z && z <= view.z_origin + h
    );
}

/* This function selects a sphere with a given radius */
int view_sphere(View view, int x, int y, int z) {
    /* param is squared radius */
    int dx = x - view.x_origin;
    int dy = y - view.y_origin;
    int dz = z - view.z_origin;
    return (dx*dx + dy*dy + dz*dz <= view.param);
}

/* views */
View view_line   = new View(input, view_select_x, 0);
View view_square = new View(input, view_cube, 5);
View view_sphere = new View(output, view_sphere, 16);
View view_cubus  = new View(output, view_cube, 3);

/* distributions */
Distribution dist_line   = new Distribution(view_line);
Distribution dist_square = new Distribution(view_square);

/* projections and net                                  origin for view */
Projection proj1 = new Projection(dist_line, view_sphere,   5,  6,  3);
Projection proj2 = new Projection(dist_square, view_cubus, 10, 10, 10);
```

Note, that we use the function `view_cube`, which selects a cubical subset of neurons, for both the presynaptic square as well as the postsynaptic cube.

The network generation process involves the following classes of objects:

- The NeuronType specifies the parameters for the learning rule and the recall method as well as some functionality, for example the transfer function.
- The Layer contains information about the size of a layer and holds the data for the neurons and their connections to the presynaptic neurons. It knows the learning rules for the connections over which its neurons receive input and the recall methods.
- The View selects a subset of neurons from a layer using an NCI6t function returning 0, if a neuron is not in the subset, and non-zero, if the neuron is in the subset. (We will later discuss an improvement.)
- The Distribution connects one postsynaptic neuron, where this neuron is determined by the Projection, with a set of presynaptic neurons.
- The Projection connects a set of postsynaptic neurons to their presynaptic neurons using a Distribution.

Specification of Layers and Neuron Types

A layer has the following specifications:

- The number of neurons. Since the neurons in the layer are arranged in a 3-dimensional grid, we will define the size of a layer by specifying the number of neurons in each coordinate direction. If e_x, e_y, e_z are the sizes in each direction and n the number of neurons in the layer, we have

  $$e_x, e_y, e_z \in \mathbb{N}, \qquad e_x > 0,\ e_y > 0,\ e_z > 0, \qquad n = e_x \cdot e_y \cdot e_z,$$

  i.e. there is at least one neuron in each layer.

- The type of neurons. This includes the transfer function, output function, temporal behavior, and the initial state.

- The type of connections from which the neurons receive input. This includes the learning rules and the learning parameters needed for these rules. It also specifies the temporal behavior of a connection.

Sometimes we want to change some parameters of a layer during the operation of a net. For this reason we will separate the specification of learning parameters and the parameters for the transfer and output functions (held in NeuronType) from the learning rule and the recall method (held in Layer). Thus we can generate more neuron types than are actually needed at one time and can so change the behavior of a layer simply by specifying a different neuron type for that layer. This leads to the following object classes:

Class: NeuronType

The basic NeuronType class is a direct subclass of Core and provides no additional functionality. The functionality must be added by further subclassing this class to, for example, a backpropagation BPNeuronType. The purpose of the basic NeuronType is to be able to provide type checking: In the Layer class specification the constructor takes a NeuronType as an argument. When the Layer class is used we can only specify instances of subclasses of NeuronType for that parameter.

Class: Layer

Class values: The following two values are set to zero by this class. This is an invalid value and must be set to the appropriate values in the subclasses:

- The size s_n of memory occupied by a single neuron.
- The size s_w of memory occupied by a single weight.

Members:

- int32 e_x, e_y, e_z: The number of neurons in the directions of the coordinates.
- NeuronType nt: The type of the neurons. This member can be replaced at any time. It is initialized in the constructor from the parameter.
- The Layer also stores the data for the neurons and their connections to presynaptic neurons, however, these are only accessible through methods.

Methods:

- Constructor(NeuronType *nt*, int32 e_x, int32 e_y, int32 e_z). These parameters are immediately saved to the corresponding object members. Then an array of neurons is allocated. The newly installed layer is registered to the network "supervisor" instance net (see page 72).

The class Layer is the basic class for specifying layers. It provides class variables holding the size of a single neuron and a single weight (i.e. the size of memory which will be occupied by the data for a neuron and a weight, respectively). Both should be multiples of 4 to avoid alignment problems on certain processors.

The basic Layer class provides functionality which is independent of the neuron model. This concerns the maintainance of the data structures holding the connections to the presynaptic neurons. When a new connection is made for a neuron in the layer, an internal Strength method (of the Layer class) is invoked. It receives the presynaptic neuron's coordinates (x, y, z) mapped into the layer's space and the neuron's coordinates (x', y', z'). The method calculates an initial weight for the connection, based on these coordinates. The Strength method of the basic Layer class, however, does not do anything, because it cannot know about a weight's semantics (i.e. type), since this is neuron model dependent.

Specification of Connections

We must specify which neurons are connected. This is done using the modularized approach outlined earlier in this chapter. It employs the classes View, Distribution, and Projection.

Class: View

Members:

- The origin o_x, o_y, o_z.
- The *layer* to which the view applies.
- The int32 *selection_function*(View *view*, int32 x, int32 y, int32 z) which returns non-zero or zero, whether the neuron belongs to the subset or not.
- An arbitrary parameter p.

Methods:

- Constructor(Layer *layer*, int32 *selection_function*(View *view*, int32 x, int32 y, int32 z), double p). These parameters are immediately saved to the corresponding members. p specifies a value that can be used by the selection function.

The view specifies a subset of neurons from the layer using the origin (o_x, o_y, o_z) as a reference point. These members are filled in by the caller. They can be used by the selection function. Sometimes the reference point is not needed, for example, when all neurons of the layer are going to be selected.

Class: Distribution

Members:

- The *view* which defines the presynaptic set of neurons.

Methods:

- Constructor(View *view*). This parameter is immediately saved to the corresponding member.

The distribution's task is to connect a set of presynaptic neurons to a single postsynaptic neuron. The set of neurons is specified by the view.

Class: Projection

Members:

- The *view* specifying the set of postsynaptic neurons.
- The *distribution* to use for the presynaptic neurons.

Methods

- Constructor(Distribution d, View v, int32 x', int32 y', int32 z'). The constructor initiates the generation of connections.

The Projection, finally, uses a view to select a subset of neurons from a layer and connects these postsynaptically to other neurons using a distribution. The coordinates (x', y', z') are the origin for the view v. The Projection must transform the coordinates of the postsynaptic neuron into coordinates feasible for the presynaptic layer. The formulas are:

$$x = \left\lfloor \frac{x' \cdot e_x}{e'_x} \right\rfloor, \qquad y = \left\lfloor \frac{y' \cdot e_y}{e'_y} \right\rfloor, \qquad z = \left\lfloor \frac{z' \cdot e_z}{e'_z} \right\rfloor, \tag{5.1}$$

where the primed variables refer to the postsynaptic layer.

5.1.2 Working with an ANN

During practical work with an Artificial Neural Network we encounter two opposing points of view:

- Sometimes we want to work with the network as a whole. For example, we want to save the state of a net and later retrieve it.
- Sometimes we want to access only parts of the network: We tell the input layer to get the input data and the output layer to place to output of a recall at some location. But most importantly, we tell a layer to perform a recall or a training step. Also investigating the state of a single layer is extremely useful.

It turns out that it is much more flexible to access the layers for network training individually. Consider the following example, which reflects typical modeling features of Geiger's ANN paradigm (Fig. 5.3): We want a network that performs pattern recognition. For this, we "overlay" the input layer with small, overlapping patches, called *receptive fields*, for feature extraction. Each receptive field can recognize a set of *features* in the hidden layer. These are parts of a pattern, such as arcs, line segments in different inclinations, crosses, dots, and the like. These features are then combined to the patterns, represented in the output layer. To train this network, we first want the receptive fields to recognize features. We have the option between "hard-coding" the features or to train the receptive fields to make them recognize the features themselves.

For the second approach—training the receptive fields—we only want to access the feature extractor of a single receptive field in this first step. When it is well-trained we

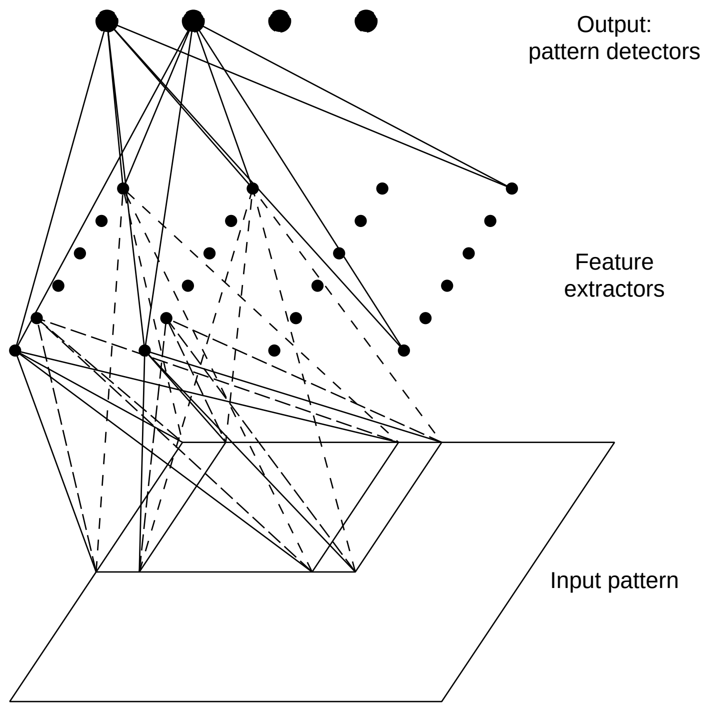


Figure 5.3: The structure of a feature extracting network, which is typical for Geiger's paradigm.

copy the weights of its connections to the other feature extractors. Then we begin to train the output layer. For this, we present input to the network and only perform a recall step in the feature layer. (Now the output at each feature extractor tells which feature was recognized.) Then the output layer performs a training step.

The process of training a layer we call a *schedule*. For the network above we need a schedule for each layer, that is, two schedules in all for this single problem. If we had only global access to the network, it would be inevitable to code the schedules into methods of a global Network class, which would involve programming in C. But this is exactly what we want to avoid. Since schedules usually change when the network does (which happens regularly during experiments), it is much more convenient to have access to the basic constituents of the net, the layers and neurons, so that the schedule can be coded in NCI6t. For this reason, the Layer class provides methods for training and recall. As a side effect, we get the possibility to perform recall steps interactively outside a schedule for experimenting purposes.

Learning and Recall

The methods for learning and doing recall are implemented with the layer class because these methods are different for each neuron model.

Class: Layer

Methods:

- void Recall(): This method performs a recall step. However, in the basic Layer class it is only a dummy, doing nothing, which must be replaced in the subclass which implements the actual neuron model.

- void Learn(): This method performs a learn step. As the Recall method, it is also a dummy, which must be replaced in the subclass.

If a neuron model allows different learning rules, it can, of course, implement further learning methods under different names.

The methods for giving input to the layers and for taking output from layers are not provided by the basic class. They must be provided by subclasses, which implement the neuron model, since these depend strongly on the learning rules, recall methods, and the semantics of the neuron model. As an example, we use a backpropagation model.

Class: BPLayer

Methods:

- void GetInput(FloatData s, int32 n, int32 o): This method places the record number n, starting with unit o, of data object s into the activity fields of the layer's neurons.
- void GetDesired(FloatData s, int32 n, int32 o): This method places the record number n, starting with unit o, of data object s into the desired output field of the layer's neurons.
- void PutOutput(FloatData d, int32 n, int32 o): This method places the values of the output fields of the layer's neurons into record number n, starting at unit o, of data object d.

Saving and Loading Network States

It is often necessary to save the state of a network to a file in order to retrieve it at a later time.

The proposed implementation of ANN in NCI6t is layer-oriented, which makes it difficult to store a whole network's state to a file. In order to make it work, each layer must register to a built-in "supervisor", which then knows about all layers. The "supervisor" provides methods for saving and loading the network state. These methods then send messages to the registered layers in some specific order, specifying an open file, which allow the layers to store or retrieve states.

This procedure saves and loads only the network state, that is, the activity and the weights of the connections. It does not save the network topology.

The built-in "supervisor" is the instance named net of the class Network. It does not make sense to create instances of this class, because the layers will always register to the built-in instance, net.

Class: Network

Members:

- string *net_name*: This member can specify a file name which contains NCI6t network generation instructions. These instructions should generate exactly the network which is currently being worked on. The name will be saved to the state file to simplify network loading.

Methods:

- void Save(string *filename*): This method saves the states of the registered layers to the specified file.
- void Load(string *filename*): This method loads the states of the registered layers from the specified file.

Loading a previously saved file will only work, if the network which loads from the file is exactly the same network which saved the file. This requires that the loading network is created in exactly the same manner as the saving network, such that the same number of neurons and the same number of connections in each layer (if not even the same connections) is generated.

This can be guaranteed, if the network is generated from a command file. For this reason, the name of this file can be specified to the member net.net_name. This value will be stored into the file specified with Save(). If the Load() method is invoked *and* there are no layers registered with net, the Load() method will first include the commands in the specified file as if an "`include`" command were issued before the net.Load() command. This will re-generate the network, which then can load the state from the file. If there are already layers registered, the name in the state file will simply be ignored. By this it is easily possible to generate the net *and* load the state with the first net.Load() and to only load a state (into the network already in memory) with the next call to net.Load().

Establishing the connection between the command file generating a network and a state is, therefore, the user's responsibility. In order to provide some security, such that it is not possible to load a state from a file which was not saved by the same network, the Save() method, as well as all layers' saving methods generate checksums and store them to the state file, allowing the reading methods to cross-check if the file was generated by the correct network. The calculation of the checksum can involve the number of neurons in the layer, the number of connections from the layer to its presynaptic layer(s), or even the indices of the neurons in the presynaptic layer(s).

Network topologies involving random numbers render this procedure difficult. Since as the network state is loaded, the network topology (at least the number of neurons and the number of connections between layers) must be exactly the same when the state was saved, it must be guaranteed that the same sequence of "random" numbers is generated every time the network is built. To achieve this, the network designer must put a "`seed`" command at the beginning of the file referenced by net.net_name.

Handling Data

A further issue in working with artificial neural networks is the handling of data. Let us reconsider the feature extracting example. We see that we have data coming from different sources: We get pictures from some frame grabbing hardware, which we use during training and recall; we get sub-pictures representing the features, which we use during feature training; for each feature we get a data record representing the desired output for that feature, which we also use during feature training; finally, for each picture of the training set we get a data record representing the desired output for that picture.

Data is usually structured. At least it is subdivided into the individual pictures. For other examples of input data each "picture" may further be divided: Consider a preprocessed image, where in addition to the image some extracted data, such as the center of gravity, the maximum extent, etc. is attached. Also, often the input data and the desired output are contained in one record.

For our implementation we assume that a data file contains a sequence of *units* of equal type, like bytes, integers, or floating point numbers. The units are arranged in *records* of equal length. If a record is logically subdivided (such as input picture and corresponding desired output in one record), the subparts are accessed by specifying a number of units as *offset* from the start of the record.

The basic Data class. This class provides functionality common to all Data classes. The record size for an instance is fixed at creation, but not the number of records.

Class: Data

Class values:

- The *unit_size* of one record element. This value is set to zero by this class. This is an invalid value and must be set to the appropriate size of the record element in the subclasses.
- The maximum number of records, *max_rec_no*, is set to 2^{14}. This value is used to avoid, that due to some error, the system would try to allocate a ridiculously large amount of memory.

Members:

- The size of a record, *rec_size*, tells how many units are contained in a single record.
- The number of records, *num_records*, tells how many records the instance currently holds.

Methods:

- void Constructor(int32 *rec_size*): The parameter tells how large the records held by the instance will be. It is recorded into the member *rec_size*. The number of records is initialized to zero.
- void Load(string *file_name*): This method allocates enough memory so that the contents of the specified file can be held. (It destroys the old contents of the instance.) Then it loads the file into that memory. A warning is generated, if the last record cannot be filled completely by the contents of the file. It is up to the user to use the last record, since part of the record may contain arbitrary data.
- void Save(string file_name): This method saves the contents of the instance to the specified file.
- void Copy(Data s, int32 f_s, int32 n_r, int32 f_d, int32 o_s, int32 n_u, int32 o_d): This method copies data. The class of s must be the same as the current instance's class. The method copies n_r records, starting with the f_s-th, from the instance s to the current instance, beginning at the f_d-th record. From every source record n_u units are copied, starting with the o_s-th unit in the source record, to the destination record, starting at the o_d-th unit. If n_u is zero, the smaller of s.rec_size $- o_s$ and this.rec_size $- o_d$ is used ("this" denotes the current instance).

- void SetMaxRecNo(int32 n): This function resets the maximum number of records the instances of the class can hold. Note, that this method resets a class value, which means that all instances of the class, not only the particular class used to invoke the method are affected.

The Load method allocates new memory large enough to hold the contents of the file. The Copy method, on the other hand, reuses already allocated memory and increases the number of records only if necessary.

The FloatData class. As an example, we specify a class which can hold floating point values.

Class: FloatData

Class value:

- The *unit_size* is initialized to 4, the size of "`float`" in NCI6t.

Methods:

- void ProcessBytes(ByteData s, int32 f_s, int32 n_r, int32 f_d, void *convert*(byte *src*[ ], float *dst*[ ])): This method is similar to the Copy method. However, the data is not copied automatically, but is done by the conversion function specified in convert. This function may take the values from the array *src*, process them in some way, and assign the result to the elements of the array *dst*. It is called once for each record. *src* refers to the whole source record, *dst* refers to the whole destination record.
- void ProcessFloats(FloatData s, int32 f_s, int32 n_r, int32 f_d, void *convert*(float *src*[ ], float *dst*[ ])): This method is analogous to ProcessBytes, except that the source is a FloatData object.

The class ByteData is similar to FloatData.

5.2 Implementation of ANN Classes

We will now present some implementation details. For this purpose a pseudo language will be used. The language is C-like. We use the arrows $\downarrow$ and $\uparrow$ to denote the direction of value-passing in parameter lists. $\downarrow$ denotes an input parameter, $\uparrow$ denotes an output parameter. When defining methods, we will use

return-type Class.Method(parameters...)

to denote that the *Method* of *Class* is being defined. In the body of the method, we use the notation "this.m" to access the member m of the instance on which the method is invoked. We use "this.class.v" to access the class value v of the class of the instance onto which the method is invoked. Variables of type NeuronSet hold sets of triples (coordinates). Variables of type File hold an internal representation of a data file. We do not further specify the type Weight because in the actual implementation it can be replaced by an anonymous pointer type (page 81).

For the sake of clarity, error checks and boundary checks are often omitted in the pseudo code.

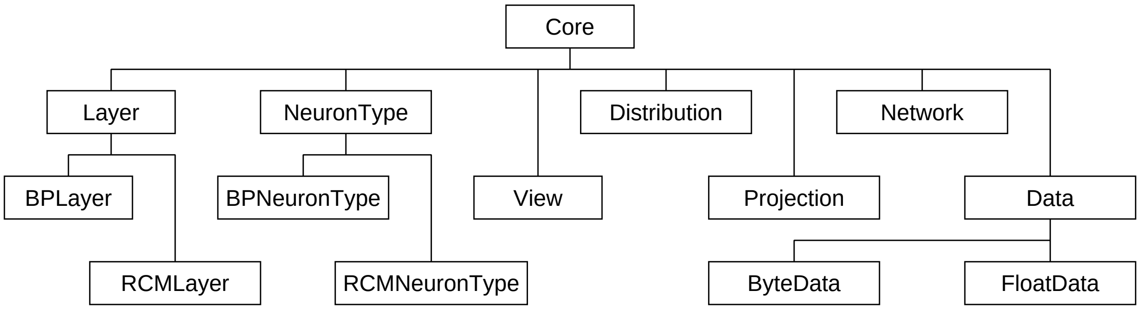


Figure 5.4: The class hierarchy showing two implemented neuron models: the back propagation model and the rate coded model.

Fig. 5.4 shows the class hierarchy implementing the ANN classes, where class A below class B (along a line) means that A is derived from B by inheritance. The figure shows the situation that two different neuron models are implemented, the backpropagation model and the rate coded model (Perceptron) [Roj93], as well as two data classes, FloatData and ByteData.

The classes for network generation, View, Distribution, and Projection, are generic, meaning that they can be used with any neuron model. The Layer and the NeuronType class, on the other hand, implement the neuron models, and must, therefore, be subclassed for each neuron model.

The basic class Layer provides methods for operating on neurons and connections. The learn and recall methods are only provided as dummies, which must be replaced in subclasses by working methods. They only serve as a pattern fixing the parameters which must be passed to these methods.

The basic class NeuronType is a complete dummy, which is used by other classes only to constrain the types of parameters and members.

The basic Data class provides methods feasible for all its subclasses, which do not need to interpret the units of data. Interpretation is only done by the subclasses.

5.2.1 Representation of Neurons and Connections

In order to make the algorithms easier to understand, we first have a look at the representation of neurons and the connections between neurons as data structures. Fig. 5.5 shows the dependencies among the structures. (Again, backpropagation serves as an example.)

The neuron data is an array of data, which is separated from the data describing the connections. Similarly, the weights are stored in an array separate from the indices of the presynaptic neurons. These separations allow that the information telling which neuron is connected with which one can be handled by the basic Layer class, because this information is independent from the neuron model. The algorithms manipulating this data need not be replicated in the subclasses. On the other hand, the structures of the neuron data and the weight do depend on the neuron model, but they can be handled generically when separated from the connection data.

For each neuron a linked list is kept. Each element in the list specifies the neurons of one layer from which this neuron receives input. Only the indices of the presynaptic neurons are stored, which is shown as dotted lines in Fig. 5.5.

As an example, a possible implementation of a backpropagation neuron is used. The

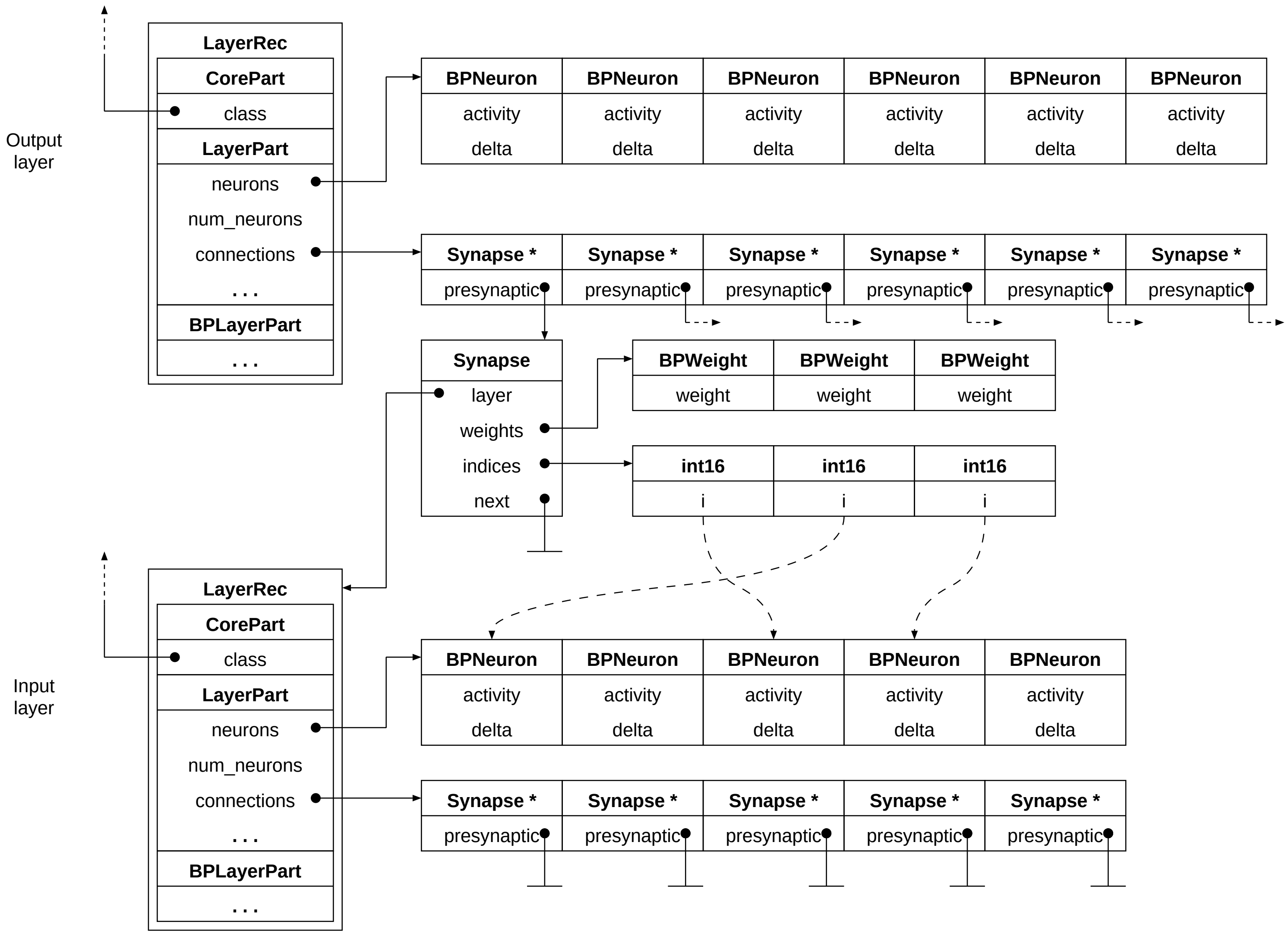


Figure 5.5: The representation of layers and connections.

basic Layer class provides the mechanism for allocating neurons and making connections. It does not provide useful recall and learning methods; the methods provided simply do nothing. The basic Layer class must be subclassed to provide these features, along with the layout of the neuron and the weight. Since these may change from neuron model to neuron model, for each model a new class must be provided.

5.2.2 Network Generating Classes

The class Projection. The constructor of the projection stores away its parameters. After this it starts generating the network:

```
void Projection.Constructor(↓Distribution d, ↓View v, ↓int32 x′, ↓int32 y′, ↓int32 z′)
{
   int32 x, y, z;
   int32 i;
   NeuronSet σ;        /* set of postsynaptic neurons */

   this.distribution := d;
   this.view := v;
   this.x_origin := x′;
   this.y_origin := y′;
   this.z_origin := z′;

   σ := this.view.GenerateSet(↓this.x_origin, ↓this.y_origin, ↓this.z_origin);

   for (x, y, z) ∈ σ {
      this.distr.Generate(↓this.view.layer, ↓x, ↓y, ↓z);
   }
}
```

The class Distribution. The constructor of the distribution stores away the specified parameter into the corresponding member.

```
void Distribution.Constructor(↓View view)
{
   this.view := view;
}
```

The internal method Generate is called by the projection to initiate the connection generation process. It receives the coordinates of the postsynaptic neuron in the space of the postsynaptic layer, which is also passed to the method.

```
void Distribution.Generate(↓Layer post, ↓int32 x′, ↓int32 y′, ↓int32 z′)
{
   int32 x, y, z;
   NeuronSet σ;        /* set of presynaptic neurons */

   Transform(↓x′, ↓y′, ↓z′, ↑x, ↑y, ↑z, ↓post, ↓this.view.layer);
   σ := this.view.GenerateSet(↓x, ↓y, ↓z);
```

```
    post.Connect(↓this.view.layer, ↓x′, ↓y′, ↓z′, ↓σ);
}
```

The function Transform maps the input-coordinates from the space of the first specified layer into the space of the second specified layer.

```
void Transform(↓int32 x′, ↓int32 y′, ↓int32 z′, ↑int32 x, ↑int32 y, ↑int32 z,
                          ↓Layer from_layer, ↓Layer to_layer)
{
    x := x′ · to_layer.e_x/from_layer.e_x;
    y := y′ · to_layer.e_y/from_layer.e_y;
    z := z′ · to_layer.e_z/from_layer.e_z;
}
```

The class View. The constructor of the view also stores away the parameters to the corresponding members. A selection function must be provided which will be invoked during the generation of a set of neurons.

```
void View.Constructor(↓Layer layer,
     ↓int32 selector(View view, ↓int32 x, ↓int32 y, ↓int32 z), ↓double p)
{
    this.layer := layer;
    this.selector_function := selector;
    this.param := p;
}
```

The internal method GenerateSet generates a set of neurons which is a subset of the layer associated with the view. The coordinates passed to the method specify the origin of the view. The view traverses all neurons in the layer and determines for each neuron if it belongs to the subset or not.

```
NeuronSet View.GenerateSet(↓int32 x, ↓int32 y, ↓int32 z)
{
    int32 i_x, i_y, i_z;
    NeuronSet σ;

    this.x_origin := x;
    this.y_origin := y;
    this.z_origin := z;

    σ := ∅;  /* empty set */
    for i_x := 0 to this.layer.e_x − 1 {
        for i_y := 0 to this.layer.e_y − 1 {
            for i_z := 0 to this.layer.e_z − 1 {
                if (this.selector_function(↓this, ↓i_x, ↓i_y, ↓i_z)) {
                    σ := σ ∪ {(i_x, i_y, i_z)};
                } /* if */
            }
        }
    }
```

```
    return σ;
}
```

The classes Layer and NeuronType. The two classes Layer and NeuronType operate closely together. After a first analysis we will find that all information concerning the neurons of a layer can be stored in a single class, Layer. This information includes the extensions in the coordinate directions, learning parameters, and the transfer functions. If a new type of neurons is needed, e.g. when the transfer function changes such that new parameters are necessary, a new class must be implemented.

During the practical work with ANN, however, it turns out that it is often desirable to change the parameters of the neurons belonging to a layer on the fly, because one wants to examine the behavior of the net with different parametrization. For this purpose, a class NeuronType is introduced whose task only is to store the parameters. The methods used by the neurons are known to the Layer class and not to the NeuronType class. By this it is easy to exchange the NeuronType associated with a class to get different behavior.

The reason why the methods are still known by the Layer and not by the NeuronType is that the Layer holds the data for the neurons in the layer, such as the current activation. If the methods where only known to the NeuronType, it were possible to exchange the NeuronType of a layer by a completely different one, for which the data held by the Layer would not be correct (if the new NeuronType needs more than the current activation, for example).

Further, we store the data for the neurons with the layer for space and time efficiency. The layer can arrange the data in a contiguous array. If the data were stored in the NeuronType, the Layer would have to store an array of NeuronType instances (one for each neuron); the creation time would be much higher because for each element space must be requested from the operating system. Every instance of NeuronType would have to be initialized with parameters. Although this would result in higher flexibility—each neuron's parameters could be changed independently—this flexibility is not needed and would result in much work to gain the effect usually wanted: changing the parameters of all neurons in one step.

The constructor first stores the parameters in corresponding members. Then it allocates enough space to hold the data for the neurons. It also tells the network supervisor about the new layer.

```
void Layer.Constructor(↓NeuronType nt, ↓int32 e_x, ↓int32 e_y, ↓int32 e_z)
{
    int32 i;

    this.e_x := e_x;
    this.e_y := e_y;
    this.e_z := e_z;
    this.nt := nt;

    /* allocate neurons */
    this.n := e_x · e_y · e_z;
    this.σ := array of this.n neurons;
    for i := 0 to this.n − 1 {
        this.σ[i].π := ∅  /* set of presynaptic neurons is empty */
```

```
    }
    net.Register(↓this);
}
```

The set this.$\sigma[i].\pi$ is a set of tuples (l, I), where l specifies the presynaptic layer and I is a set of tuples (j, w) indicating the subset of neurons in the presynaptic layer from which the neuron this.$\sigma[i]$ receives input (j indexes the array of neurons in the presynaptic layer and w is the weight of the connection).

When the network is generated, the Distribution calls the postsynaptic layer's Connect method.

```
void Layer.Connect(↓Layer pre, ↓int32 x′, ↓int32 y′, ↓int32 z′, ↓NeuronSet σ)
{
    int32 x, y, z;
    int32 i_x, i_y, i_z;
    Weight w;

    I := ∅;
    for (i_x, i_y, i_z) ∈ σ {
        Transform(↓i_x, ↓i_y, ↓i_z, ↑x, ↑y, ↑z, ↓pre, ↓this);
        j := (index of neuron at i_x, i_y, i_z in layer pre);
        this.Strength(↓x, ↓y, ↓z, ↓x′, ↓y′, ↓z′, ↑w);
        I := I ∪ {(j, w)};
    }
    this.σ[i].π := this.σ[i].π ∪ {(pre, I)};
}
```

i is the index of the neuron which is going to be connected with the neurons in σ of the presynaptic layer specified by pre.

For every neuron model a new subclass of Layer must be implemented along with a new subclass of NeuronType, because the parameters for the new type of neurons usually change. The type of w, the weight of the connection to the presynaptic neuron, also depends on the neuron type.

This method uses the internal method Strength, which calculates an initial weight for the connection. It can use the coordinates of the presynaptic neuron (x, y, z) and the postsynaptic neuron (x', y', z'). The basic Layer's Strength method cannot calculate a weight since the semantics (i.e. the type) of the weight depends on the neuron model. The value w returned by the basic Layer's Strength method is undetermined. The Strength method must be overloaded in subclasses of Layer. For example, in BPLayer the Strength method will calculate small random weights.

Efficiency improvements

Let us now make some efficiency considerations. From the algorithms above we see that for both the presynaptic and the postsynaptic layer a view is evaluated. We consider now a network (or part of a network) with two equally large layers with 128×128 neurons, where every postsynaptic neuron is connected with exactly one presynaptic neuron, the one with the same location within the layer (recall that the neurons of a layer are arranged on a regular grid determining the coordinates of each neuron). Such networks are frequently parts of larger networks used for image analysis.

During the generation of the ANN for every postsynaptic neuron the whole presynaptic layer is scanned (in the presynaptic view) to find that specific neuron which shall be connected to the postsynaptic neuron. We find that for this the presynaptic selection function is called

$$128 \cdot 128 \cdot 128 \cdot 128 = 268\,435\,456$$

times in order to establish the

$$128 \cdot 128 = 16\,384$$

connections—a vast effort, which can be avoided. The reason for that many calls to the selection function is that all neurons of the layer are tested in order to *select* the ones we are interested in.

A different method to find the interesting neurons would be to *produce* this set. This implementation of the class View would then be:

```
void View.Constructor(↓Layer layer, ↓int32 func(View view))
{
   this.layer := layer;
   this.view_function := func;
}
NeuronSet View.GenerateSet(↓int32 x, ↓int32 y, ↓int32 z)
{
   this.x_origin := x;
   this.y_origin := y;
   this.z_origin := z;

   this.σ := ∅;  /* empty set */
   this.view_function(↓this);
   return this.σ;
}
```

The view_function can use the method AddToSet() to include a neuron at a specified coordinate into the set:

```
void View.AddToSet(↓int32 x, ↓int32 y, ↓int32 z)
{
   this.σ := this.σ ∪ {(x, y, z)};
}
```

For example, the view_function selecting a cube given the center can be:

```
void view_cube(View view)
{
    /* param is length of side; origin is center */
    int32 h = view.param / 2;
    int32 xl = view.x_origin - h;
    int32 xh = view.x_origin + h;
    int32 yl = view.y_origin - h;
    int32 yh = view.y_origin + h;
    int32 zl = view.z_origin - h;
    int32 zh = view.z_origin + h;
```

```
    for (ix = xl; ix <= xh; ix += 1) {
        for (iy = yl; iy <= yh; iy += 1) {
            for (iz = zl; iz <= zh; iz += 1) {
               view.AddToSet(ix, iy, iz);
            }
        }
    }
}
```

(Note, that this is only a primitive implementation which lacks accuracy and security checks.) In this example we see that it is possible to implement a “selection” function that only considers those neurons which actually will be connected. This will reduce the time required for the generation of a network considerably. Sometimes it may, however, still be necessary to consider more neurons (or even every neuron). But this must now be explicitly programmed in the view function.

5.2.3 Methods for Scheduling

We now discuss the methods necessary for writing a schedule.

Methods of the Layer

The layer must provide a recall method and a learning method for each learning rule which is applicable to the neuron model the layer supports. These methods in turn need methods which calculate the output and the transfer function of a neuron.

The Recall method. The recall method's task is to calculate the new activities of the neurons. The actual implementation of the recall method depends, of course, on the type of neurons which are used in the layer. Most likely, however, it looks like the recall method of the backpropagation neuron:

```
void BPLayer.Recall()
{
   int16 j;
   double input;

   for j := 0 to this.n − 1 {
      input := 0;
      for (l, I) ∈ this.σ[j].π {        /* set of presynaptic neurons */
         for (i, w) ∈ I {
            input += l.Output(↓i) · w;
         }
      }
      this.neurons[j].activity := this.Transfer(↓j, ↓input);
   }
}
```

This code is general enough to allow that a neuron receives input from neurons of different layers. In such cases the set this.$\sigma[j].\pi$ contains more than one element (one for each layer from which input is received).

The Learn method. Since the learning in a network strongly depends on the type of neurons, a general learning method cannot be stated here. The task of a learning method is to update the weights of all connections from which a neuron receives input taking into account its current state, and, in the case of supervised learning, some supplied values.

If the learning rule requires that before adjusting the weights a recall step is performed, it is probably wise to *not* hard-code this in the Learn method. It is more flexible if it is assumed that this recall step is explicitly invoked by the operator.

The Transfer method. The Layer's Transfer method is called by the Recall and (possibly) by the Learn methods. It gets to know which neuron is currently being worked on:

```
double BPLayer.Transfer(↓int16 i, ↓double x)
{
   return this.nt.Transfer(↓x);
}
```

The method can, of course, ignore this information, as is shown above for the Transfer method of a backpropagation neuron, since the transfer function does not depend on the state of the neuron; it is only interested in the input x the neuron receives.

As the example shows, the Layer class's Transfer method simply depends on the neuron type's transfer method. By this, the layer can, for example, use an exponential sigmoid in one run and an arc-tangent function in the next by exchanging the NeuronType of the layer.

The Output method. The Layer class must provide an Output method which allows the Recall and Learn methods to query the output of a neuron:

```
double BPLayer.Output(↓int16 i)
{
   return this.nt.Output(↓this.neurons[i].activity);
}
```

The Output method only gets to know which neuron is currently being worked on. It usually depends on the neuron type's Output method.

Input/Output methods. For getting input data from a data object and putting output to a data object, it is necessary to know both the data stored in a neuron as well as the type of data that the data object stores. For this reason, the data exchange methods can only be implemented in subclasses.

The implementations of the backpropagation's GetInput, GetDesired, and PutOutput methods are very similar. The main difference is where the neuron's data is accessed.

```
void BPLayer.GetInput(↓FloatData s, ↓int32 n, ↓int32 o)
{
   Pointer p;          /* pointer to the source */
   int32 r;            /* the number of units available in the source */

   s.GetPointer(↓n, ↓o, ↓"don't extend", ↑p, ↑r);
   if (r < this.n) {
```

```
        [Error, too few values in source]
    }
    for i := 0 to this.n − 1 {
        this.neurons[i].activity := [floating point value at p];
        [increase p by the size of a floating point value]
    }
}

void BPLayer.GetDesired(↓FloatData s, ↓int32 n, ↓int32 o)
{
    Pointer p;          /* pointer to the source */
    int32 r;            /* the number of units available in the source */

    s.GetPointer(↓n, ↓o, ↓"don't extend", ↑p, ↑r);
    if (r < this.n) {
        [Error, too few values in source]
    }
    for i := 0 to this.n − 1 {
        this.neurons[i].desired := [floating point value at p];
        [increase p by the size of a floating point value]
    }
}

void BPLayer.PutOutput(↓FloatData d, ↓int32 n, ↓int32 o)
{
    Pointer p;          /* pointer to the destination */
    int32 r;            /* the number of units available in the destination */

    d.GetPointer(↓n, ↓o, ↓"extend", ↑p, ↑r);
    if (r < this.n) {
        [Error, too few values in destination]
    }
    for i := 0 to this.n − 1 {
        [floating point value at p] := this.neurons[i].activity;
        [increase p by the size of a floating point value]
    }
}
```

Methods of the NeuronType

The main purpose of the NeuronType class is to define the behavior of a neuron. For this, a neuron's transfer and output functions are implemented as methods of that class. The members of a NeuronType are the parameters needed by these methods.

The goal of the separation of the NeuronType from the layer is that the behavior of all neurons of a layer can be changed at once. This gives us the flexibility which is necessary when we want to study the behavior of the same network structure with different parametrizations, and even different transfer functions.

The Transfer method. A neuron usually modifies the input it receives by a transfer function. In the implementation the transfer function is provided by a Transfer method in the NeuronType class. The transfer function usually depends on some parameters; for example, the linear threshold needs a lower bound and an upper bound. These parameters are stored as members of the class. As an example we show the implementation of a sigmoid (assuming we have a class BPNeuronType):

double BPNeuronType.Transfer($\downarrow$double x)
{
  return $1/(1+\exp(-x))$;
}

The Output method. The Layer must provide a method which calculates the output of a neuron, taking into account its current state. Similar to the Transfer method, the Output method can be parametrized taking values from the NeuronType associated with the class. The output function often only is the identity function.

double BPNeuronType.Output($\downarrow$double x)
{
  return x;
}

Efficiency Considerations

From the algorithm of the Recall method we see that for every connection the Output method of the presynaptic layer is called. In many applications, however, the output is simply the activation of the presynaptic neuron. Furthermore, the neuron model in the postsynaptic layer is frequently the same as in the presynaptic layer. In these cases, the (postsynaptic) Recall method could simply access the activation value of the presynaptic neuron, since it knows that neuron's data structure (because it is the same as its own). In Fig. 5.5, for example, the type of the neurons is BPNeuron in both the input and the output layer.

For this to work properly, the Connect method of a layer must make sure that the class of the presynaptic layer matches its own class (using the NciIsSubclass function).

5.2.4 Handling Data

For network training and recall we need data. We assume that, whatever the physical source of this data is, it will eventually be available in a file, which can be read by the application. To provide data to the learning and recall methods we first read it into memory, where it is maintained by objects of class Data and its subclasses.

The data handled by these objects is assumed to be arranged in records of equal size. Each record consists of a number of units of one specific type. The basic Data class provides functionality which is independent of the actual data type of a unit (as long as its size is known). The subclasses of this class must be implemented whenever the methods need to know about the type of the unit.

The Data class.

The basic class Data provides functionality which is independent of the actual data type of the units of data that are being processed.

The Constructor and the Destructor. The record size for a data object is already fixed at creation time. The parameter rec_size specifies the number of units which will be contained in one record.

```
void Data.Constructor(↓int32 rec_size)
{
   if (this.class.unit_size ≤ 0) {
      [Error; unknown unit size]
   }
   this.rec_size := rec_size;
   this.num_records := 0;
   this.data := allocate(↓0);     /* a dummy, so reallocation works */
}
```

This implementation shows how run-time security checks can be done. The particular check above makes sure that the instance, which is created, has properly set up the unit size—it must be greater than 0. Instantiating the basic class Data will always fail, because its unit_size *is* initialized to 0. The reason for this is that the data held by a data object *must* have an interpretation (i.e. a type) of its units, so it *must* be subclassed.

```
void Data.Destructor()
{
   free(↓this.data);
}
```

The GetPointer method. This internal method calculates a pointer to data given the record number and an offset within the record. It also returns the number of units which may be read or written, beginning at the location pointed to by the pointer, to stay within the record boundaries. A flag indicates if the method shall extend the memory used by the Data object if the record number is higher than the number of records held by the object, or if an error shall be issued.

```
void Data.GetPointer(↓int32 n, ↓int32 o, ↓int32 e, ↑Pointer p, ↑int32 r)
{
   if (n ≥ this.num_records and e = "extend" and n ≤ this.class.max_rec_no) {
      this.data := reallocate(↓this.data, ↓(n + 1) · this.rec_size · this.class.unit_size);
   } else {
      [Error, record number out of range]
      p := 0; r := 0;
      return;
   }
   p := (n · this.rec_size + o) · this.class.unit_size;
   r := this.rec_size − o;
}
```

The Load and Save methods. These methods treat the file contents as *binary*, which means that no conversion takes places as the file is read or written. By this, they are generic and need not be re-written for each unit type.

The Load method destroys the old contents of the data object (if present). It allocates enough memory that can contain the whole file. If the file is too short, so that it does not completely fill the last record, a warning is generated.

void Data.Load($\downarrow$string file_name)
{
  int32 n;   /* the number of records in the file */
  int32 s;   /* the size of a record in bytes */
  int32 f;   /* the file */
  int32 t;   /* the number of bytes in the file */

  s := this.class.unit_size $\cdot$ this.rec_size;
  f := fopen($\downarrow$file_name, $\downarrow$"r");
  [Find size t in bytes of file f]
  $n := (t + s - 1)/s$;
  if ($t \neq n \cdot s$) {
    [Warning, incomplete record]
  }
  this.data := reallocate($\downarrow$this.data, $\downarrow n \cdot s$);
  [read t bytes from file f into memory pointed to by this.data]
  fclose($\downarrow f$);
  this.num_records := n;
}

The Save method simply writes the chunk of memory holding the data records to the specified file.

void Data.Save($\downarrow$string file_name)
{
  f := fopen($\downarrow$file_name, $\downarrow$"w");
  [write (this.class.unit_size $\cdot$ this.rec_size $\cdot$ this.num_records) bytes to file f from the memory pointed to by this.data]
  fclose($\downarrow f$);
}

The Copy method. This method copies records from a Data object, thereby selecting parts of records. The source is the object specified as parameter, the destination is the current object. The source and destination objects must be of the same class.

void Data.Copy($\downarrow$Data s, $\downarrow$int32 f_s, $\downarrow$int32 n_r, $\downarrow$int32 f_d, $\downarrow$int32 o_s, $\downarrow$int32 n_u, $\downarrow$int32 o_d)
{
  int32 r_s, r_d;   /* remaining units in source and destination */
  Pointer p_s, p_d;   /* data pointers into source and destination */
  int32 n;   /* number of records to copy */
  int32 m;   /* no more than that many units will be copied */
  int32 s;   /* number of bytes per record to copy */
  int32 i_s, i_d;   /* record number currently being copied */

```
    if (this.class ≠ s.class) {
        [Error, source class doesn't match destination class]
        return;
    }
    n := min(↓n_r, ↓s.num_records − f_s);
    m := min(↓s.rec_size − o_s, ↓this.rec_size − o_d);
    if (n_u = 0) {
        s := this.class.unit_size · m;
    } else {
        s := this.class.unit_size · min(↓n_u, ↓m);
    }
    i_s := f_s + n;
    i_d := f_d + n
    for i := 1 to n {
        i_s := i_s − 1;
        i_d := i_d − 1;
        s.GetPointer(↓i_s, ↓o_s, ↓"don't extend", ↑p_s, ↑r_s);
        this.GetPointer(↓i_d, ↓o_d, ↓"extend", ↑p_d, ↑r_d);
        [Copy s bytes from p_s to p_d]
    }
}
```

We begin copying from the end so that the destination memory will be extended at most once. We do not use the values returned in r_s and r_d.

The SetMaxRecNum method. This method resets the maximum number of records a class can hold. This value is used to avoid that due to some error during the usage of the data objects the system would try to allocate an enormous amount of memory. Since the default maximum number of records is $2^{14} = 16\,384$, this method will only be used under rare circumstances.

```
void Data.SetMaxRecNum(↓int32 n)
{
    this.class.max_rec_num := n;
}
```

As this method resets a class value, all instances of the current object's class are affected by a call to the method, not just the current instance.

5.2.5 Methods for Saving and Loading Network States

Methods of the "supervisor"

The network supervisor class Network provides methods for saving and loading network states. The layers register with the built-in instance net, so it can advise the layers to save or load states.

The Register method. This method is invoked by the layer instances when they are created.

```
void Network.Register(↓Layer l)
{
   [add l to the list of layers in this]
}
```

The actual code for this will be some linear list maintainance. If this list is traversed later, it will give the layers in the order they registered.

The Checksum method. The Network class provides a Checksum method which calculates a value that can help to verify that two networks have the same structure. It is guaranteed that the checksums of two networks will be equal if the network structure is equal. The Network class's method uses the checksums of the layers registered with the network to get a checksum for the whole network.

The Save method. The Save method opens a file for writing. Then it writes network specific data. After this the list of layers is traversed and each layer is sent a Save message:

```
void Network.Save(↓string file_name)
{
   File f;
   Layer l;

   f := fopen(↓file_name, ↓"w");
   if (this.net_name ≠ "") {
      write(↓f, ↓"net_name=%s\n", ↓this.net_name);
   }
   write(↓f, ↓"checksum=%d\n", ↓this.Checksum());
   for l in list of layers in this {
      l.Save(↓f);
   }
   fclose(↓f);
}
```

The Load method. The Load method first checks if there are layers registered. If not, then the network specified in the state file must first be generated by including the commands from the network file. If there are already layers registered, this step is simply skipped. The state is then loaded into the layers and connections, thereby checking the checksums for correctness.

```
void Network.Load(↓string file_name)
{
   File f;
   int32 c;
   string name;

   f := fopen(↓file_name, ↓"r");
   scan(↓f, ↓"net_name=%s\n", ↑name);
   scan(↓f, ↓"checksum=%d\n", ↑c);
   if (list of layers in this is empty) {
```

```
        NciInclude(↓name);
        this.net_name := name;
    }
    if (c ≠ this.Checksum()) {
        [Error, mismatching checksums]
        return;
    }
    for l in list of layers in this {
        l.Load(↓f);
    }
    fclose(↓f);
}
```

Methods of the Layer

The basic Layer class provides a Save and a Load method which only handle the checksums, they cannot save neuron activities and network weights because the basic class does not provide this functionality. These methods are intended to be replaced in subclasses, but can be invoked by the replacement methods to simplify them, which then need not care about checksums anymore.

The Checksum method. The class Layer provides a Checksum method which is independent of the neuron model. It only uses the network structure to calculate the checksum, i.e. the number of neurons in each coordinate direction, the total number of neurons, the number of connections per neuron, and the index of the presynaptic neuron at each connection.

The Save method. The basic Save method is fairly simple.

```
void Layer.Save(↓File f)
{
    write(↓f, ↓"checksum=%d\n", ↓this.Checksum());
}
```

The Load method. Since this method is symmetric to the load method, it is equally simple.

```
void Layer.Load(↓File f)
{
    int32 c;

    scan(↓f, ↓"checksum=%d\n", ↑c);
    if (c ≠ this.Checksum()) {
        [Error, mismatching checksums]
    }
}
```

The Load method compares the checksum stored in the state file with the checksum expected by the layer and issues an error if they mismatch.

Chapter 6

The Integrated Simulation Tool at Work: A Prototype

Now we come back to the original starting point of this work. Having specified the new implementation approach NCI6t, we are discussing below a prototype of the integrated artificial neural networks and robotics simulation environment which has been implemented already. However, since there does not yet exist a sufficiently complete implementation of NCI6t, the resulting prototype program is based on NETUSE, version 4.0 (Chapter 2) (since this software is easy accessible for us), and the SAVE system in the version of autumn 1992 (Chapter 3).[1] It is of course in the spirit of our work to build a future tool using our new concept as developed in this thesis.

Scheme of training. We briefly describe the general scheme how an ANN is trained to represent the inverse kinematics of a robotic manipulator and how the resulting network is used by the robot simulator. The structure and functionality of the integrated tool is schematically illustrated in Fig. 6.1. In our application we will concentrate on the problem of inverse kinematics computation. Recall that the robot simulator represents the robot map $\mathsf{R} : \mathcal{C} \rightarrow \mathcal{W}$, which maps a joint coordinate $\mathbf{q}$ to the workspace coordinate $\mathbf{x} = \mathsf{R}(\mathbf{q})$, i.e. the direct kinematics model. The idea of our approach is to represent the inverse kinematics model under discussion by learning it with a selected neural network using suitable training data generated by our robot simulator.

First of all, an adequate training set must be found. For this, a set of joint coordinate tuples is selected. Now for each joint coordinate tuple $\mathbf{q}_i$ the corresponding workspace coordinate tuple $\mathbf{x}_i$ (endeffector position) is calculated using the robot simulator.

After this, training can begin, where the workspace coordinate tuples $\mathbf{x}_i$ are presented as input and the jointspace coordinate tuples $\mathbf{q}_i$ as corresponding desired output to a selected neural network (application of the ANN simulator).

When the network is successfully trained, that ANN can be considered as an implicit representation of the inverse kinematic model.

Now we are going to test this representation. A desired endeffector position is given as input to the network. After a recall step, the output of the ANN will produce an "actual" jointspace coordinate tuple. This is now given as input to the robot simulator (robot map), which calculates the corresponding endeffector position to that input. In the

[1] Whereas the SAVE system is said to be finished, NETUSE has meanwhile gone into version 5.0 (NEUROtools).

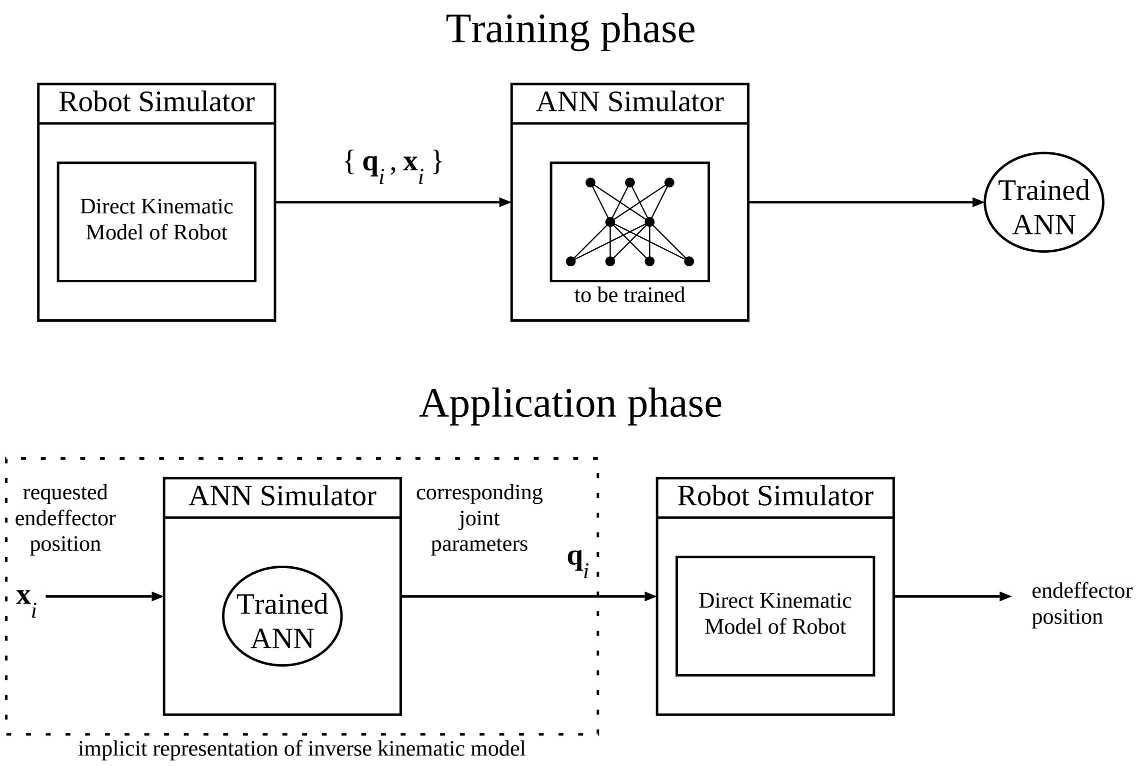


Figure 6.1: The scheme for training and recall.

Table 6.1: The joint values used for training. Each input sample is a combination of 4 values (one from each line).

No.	Joins	Type	Values
1	Base and Shoulder joints	rotational	$-\pi/4$, $-\pi/8$, 0, $\pi/8$, $\pi/4$
2	Should and Arm joints	rotational	0.0, 0.6, 1.2, 1.8, 2.4, 3.0
3	Arm and Hand joints	translational	-5000, -3500, -2000, -500, 1000
4	Hand and Tool joints	rotational	$-\pi$, $-\pi/2$, 0, $\pi/2$, π

ideal case, this endeffector position shall match the desired position originally showed to the ANN. In practice we obtain deviations from this ideal case, that means the network represents only an approximate inverse kinematics model of the robot.

One experiment was tried out with the prototype program to get a feeling for the capabilities and the performance that can be expected. A simple network was trained with the inverse kinematics of the Scara robot.

Training set. A set of $5^4 = 625$ points was chosen as training set. Each sample consisted of a combination of the values of Table 6.1 (one value from each line). These values describe a hypercube in jointspace coordinates. For each input sample the according workspace coordinates were calculated using the forward kinematics module of the robot simulator. Each workspace coordinate tuple consists of 12 values (the 3×3-orientation matrix and a position vector).

Test set. For testing a set of jointspace coordinate tuples is used, which when fed to the robot (in sequence) results in the path of the endeffector as shown in Fig. 6.4. (The endeffector is also rotated as the hand moves along the path—this cannot be visualized in a static picture.)

Coding. All of the values that are fed to the net and are going to be produced by it are floating point numbers. Since the network which we would use only allows integral neuron activities and connection weights the numbers were encoded using the "topological encoding method" introduced in [Kin90], which we briefly describe below.

Fig. 6.2 shows how the value -3100 is coded into the activities of 8 neurons. We assume that values in the range -5000 to 1000 must be encoded. 8 neurons are "positioned" at equal intervals from -5500 to 1500. (A margin of 500 is left so that the values at the limits are coded more exactly.) A Gaussian curve is put over the range such that its peak is at the value $\mu = -3100$ to be coded. The neuron at "position" x receives the activity

$$255 \cdot \mathrm{e}^{-(x-\mu)^2/\sigma^2}$$

(rounded to the nearest integer) where σ is chosen such that at most 3 neurons get "large" activities for any value of μ in the coding range (e.g. $\sigma = 600$). Therefore, -3100 is coded into the sequence of activities 0, 1, 164, 94, 0, 0, 0, 0.

Furthermore, the angle values were not used directly. Each was further coded in two values, the sine and the cosine of the angle. This avoids the discontinuity at π and $-\pi$. When the network produces the two values, they are considered as the coordinates of a point (in the plane) which determines the resulting angle.

Fig. 6.3 gives a typical set of activities (255, 150, 0, 0, 255, 255, 30, 200) which the output neurons of a network produce after a recall step. These activities encode a floating point value, which is decoded by taking the "center of gravity":

$$\frac{\begin{array}{l}-255 \cdot 5500 - 150 \cdot 4500 - 0 \cdot 3500 - 0 \cdot 2500 - \\ \quad - 255 \cdot 1500 - 255 \cdot 500 + 30 \cdot 500 + 200 \cdot 1500\end{array}}{255 + 150 + 0 + 0 + 255 + 255 + 30 + 200} \approx -1984.716.$$

Although during training a Gaussian curve is presented as desired output to these neurons, the actual output is usually not a Gaussian at all. This does not matter as long as the value decoded from the output activities corresponds to the desired value.

The network. For the example, a 2-layered, fully connected network was used. Since there were 12 input values (workspace coordinates) and 7 output values (3 angles per sine and cosine, 1 distance), each of which is coded into 8 neurons. The resulting network had 96 input neurons and 56 output neurons. Each neuron can have an integral activity in the range 0 to 255. Input and output layers were fully connected. The weights were integers in the range -32768 to 32767.

Forced Learning was used for training. This is a supervised learning method which works as follows: Consider a connection with weight w having the presynaptic actual activity o, the postsynaptic actual activity a, and the postsynaptic desired activity d.

- If $|a - d| < \epsilon$, do not change the weight of the connection (for some small ϵ, e.g. $\epsilon = 5$).
- If $o = 0$, do not change the weight or change it slightly towards 0.

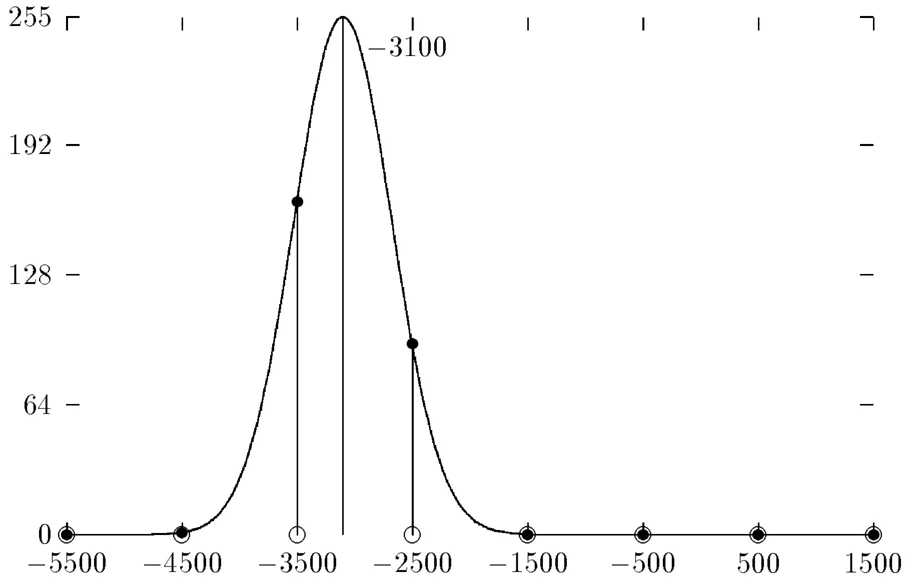


Figure 6.2: Topological encoding. These neuron activities encode the value −3100.

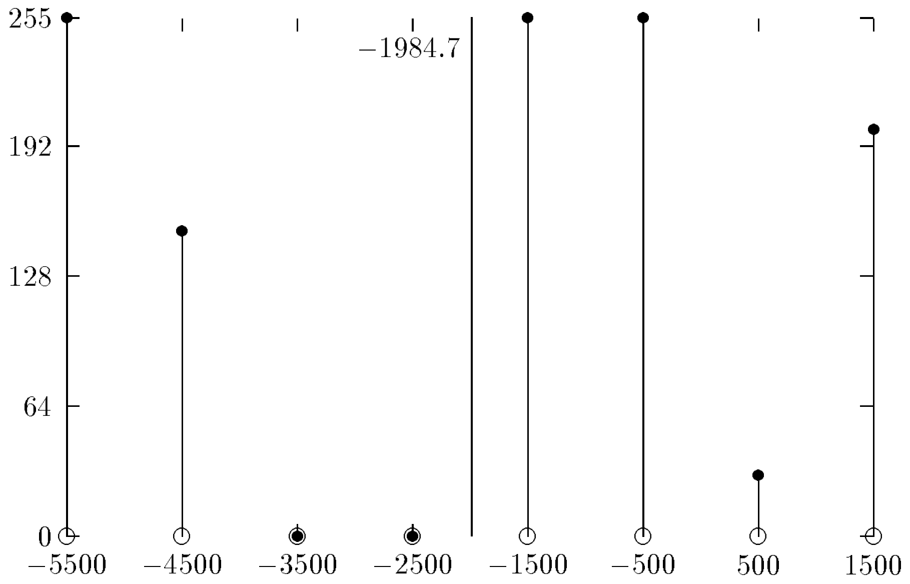


Figure 6.3: Topological decoding. The "center of gravity" of the neuron activities defines the value they encode.

- Otherwise, if $a > d$ then change the weight by a small negative amount c_- (e.g. $c_- = -5$); otherwise, ($a < d$) change the weight by a small positive amount c_+ (e.g. $c_+ = 7$). c_+ and c_- are relatively prime so that (theoretically) every integral weight can be reached.

The transfer function in the output neuron was a linear threshold (Fig. 2.2) with $a = 255$, $\theta_l = 0$, $\theta_u = 40000$.

Results. The network was trained for 100 complete runs through the training set. Statistics during the training period showed that the network would not converge. Fig. 6.5 shows the network at work. The details of the test path could not be replicated by the trained network. Even if the training was extended to 600 cycles, no further improvement could be observed. Notice, however, that the output is not purely random. We see that the overall-movement resembles the desired path qualitatively. In particular, the up-down-movement of the Hand was recognized by the network. Nevertheless, the result may already be useful in practice, since the fine-tuning of the endeffector can be done by sensors, as engineers would argue.

The experiment was the first brute-force, straight-forward approach to the problem (with the only goal to test the prototype). There are a bunch of obvious improvements which remain to be tried out, as for example:

- During training, a learning step was performed for every training sample. Learning steps can be reduced to only those cases where the desired output value and the actual output value (the un-encoded values, *not* the activities of the neurons) differ too much.

- The currently used encoding using the Gaussian is not a bijective encoding. A different method can be used.

- The training set may be too small. It could be extended to some thousands of samples. This will also increase the time needed for training proportionally.

- Longer training can be tried out.

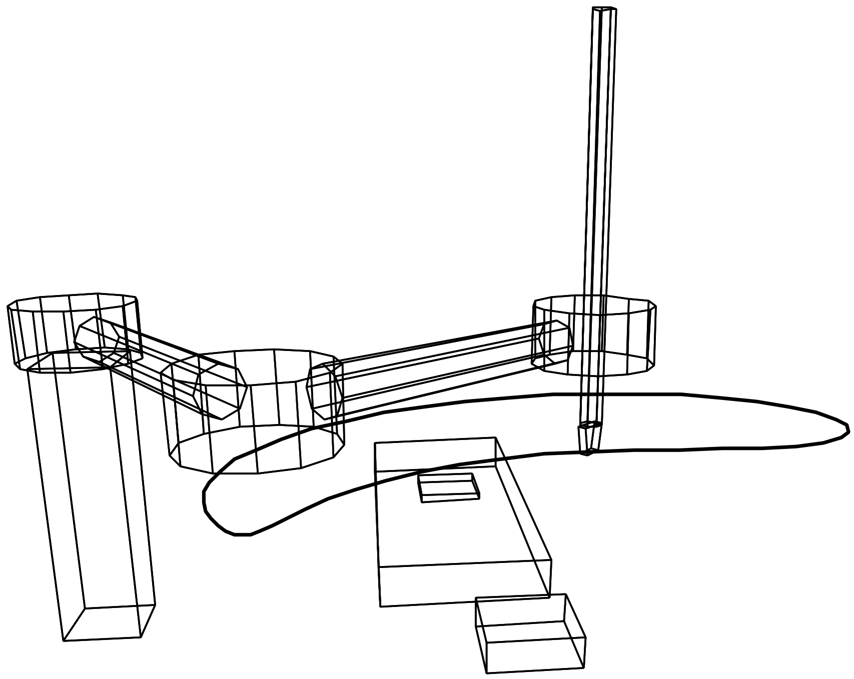


Figure 6.4: The desired test path.

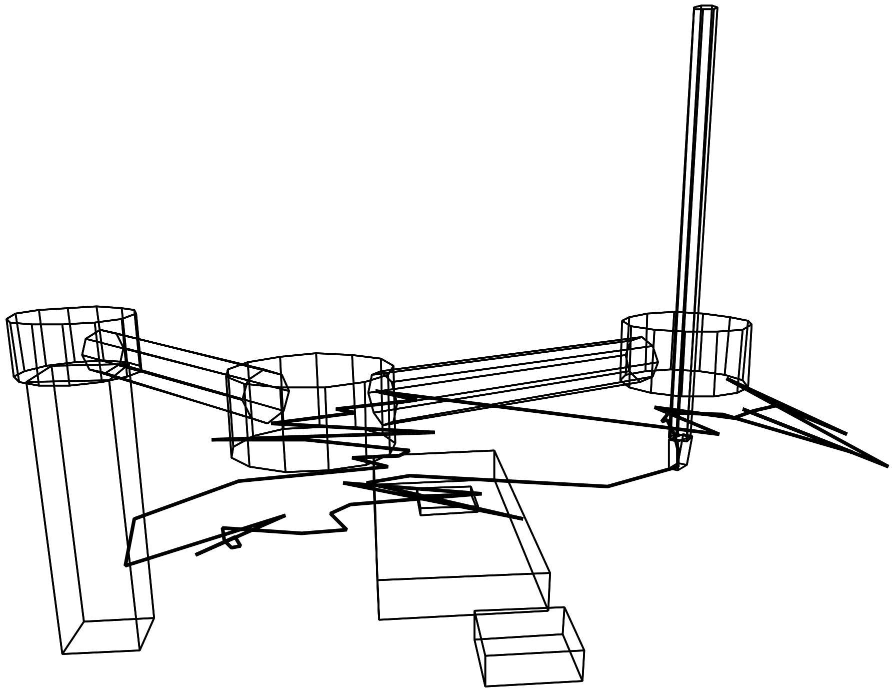


Figure 6.5: The path produced by the network after 100 training cycles.

Chapter 7

Conclusions

Based on some observations in working with an already existing ANN simulation tool, we identified some drawbacks in the user interface which we wanted to eliminate. This was the background motivation for this thesis. The new work we are presenting is a completely new design of a general command interface suitable for various purposes. Instead of a simple command dispatching mechanism, a full language is proposed. This language features all control constructs known from programming languages (like loops, selections, assignments, sub-routines, expressions, functions), strictly typed variables as well as language constructs to work in a restricted, object oriented fashion. The language has a programming interface to the language C, so that it can be extended with functionality to fit special purpose applications.

A language extension is specified using the new user interface which allows to generate artificial neural networks and to work with them. For this purpose, the object oriented features of the language are exploited. This strongly supports the reusability of already written code (generic programming techniques). This part of work definitely forms the new ANN simulator. Our concept design is flexible enough to allow the definition of natural interfaces to specific symbolic computation tools.

In a prototype demo-example using the available ANN and robotics simulation programs, we demonstrate the principle how such an integrated simulation tool can be applied to a concrete problem.

For the nearest future it is planned to complete the implementation of the new command language. Then the integrated simulation tool can be implemented based on the new system using the ANN simulator which was proposed.

The next step is to systematically study the application of ANN methods to hard robotics problems. The requirements for this research motivated the usage of Geiger's network paradigm as well as an "open" implementation of the command language (and the whole tool), so that extensions to the existing software can be established with least effort.

Planned extensions are interfaces which allow to use selected symbolic computation software as well as fuzzy logic methods and genetic algorithm approaches, which is part of work in the ANNig at RISC, especially by Markus Meisinger (genetic algorithms) and Thomas Hainzl (fuzzy methods).

Literature

[Ang88] Jorge Angeles. *Rational Kinematics.* Springer, New York-Berlin, 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[Bur89] J. W. Burdick. On the inverse kinematics of redundant manipulators: Characterization of the self-motion manifolds. *IEEE*, pages 264–270, 1989.

[CP89] Victor C. Chen and Yoh-Han Pao. Learning Control with Neural Networks. In Hollerbach [Hol89], pages 1448–1453.

[DH55] J. Denavit and R. S. Hartenberg. A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. *ASME Journal of Applied Mechanics*, pages 215–221, 1955.

[DH86] Rüdiger Dillmann and Martin Huck. A Software System for the Simulation of Robot Based Manufacturing Processes. *Robotics*, 2:3–18, 1986.

[DPSS91] Fatiam Dargam, Jochen Pfalzgraf, Volker Stahl, and Karel Stokkermans. Towards a Toolkit for Benchmark Scenarios in Robot Multi-tasking. Technical Report 91-45.0, RISC-Linz, University of Linz, Austria, 1991.

[GP94] Hans Geiger and Jochen Pfalzgraf. On a Network Paradigm and its Mathematical Model, to appear in 1994.

[Gro88] Stephen Grossberg. Nonlinear Neural Networks: Principles, Mechanisms, and Architectures. *Neural Networks*, 1(1):17–62, 1988.

[Hai94] Thomas Hainzl. A Report on Fuzzy Sets and Neural Networks (a Formal Introduction). Technical Report no. 94-15, RISC-Linz, University of Linz, Austria, 1994.

[Heb49] Donald O. Hebb. *The Organization of Behavior.* Wiley, New York, 1949.

[Heb88] Donald O. Hebb. The Organization of Behavior. In *Neurocomputing: Foundations of Research*, pages 43–56. MIT Press, Cambridge, 1988. Reprint of [Heb49], Introduction and Chapter 4.

[Hei85] Hermann Heiß. *Die explizite Lösung der kinematischen Gleichung für eine Klasse von Industrierobotern.* PhD thesis, Technische Universität München, April 1985.

[HJV90] Bill Horne, M. Jamshidi, and Nader Vadiee. Neural Networks in Robotics: A Survey. *Journal of Intelligent and Robotic Systems*, 3(1):51–66, 1990.

[Hoc89] Harald Hochmann. *Programmieren mit Clipper.* Markt & Technik, Haar bei München, 1989.

[Hol89] John M. Hollerbach, editor. *Proceedings of the IEEE International Conference on Robotics and Automation.* IEEE Computer Society Press, Washington, 1989.

[IEE87] IEEE. *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, 1987.

[ISO87] International Standardization Organization ISO. *Information Processing Systems—Computer Graphics—Programmer's Hierarchical Interactive Graphics System (PHIGS). Functional Description.* ISO/IEC/DIS, 1987.

[Kar88] U. Karras. On Mathematics in Robotics. Lecture Notes (in German), DMV Seminar on Mathematics in Robotics; held in Blaubeuren, November, 1988.

[Kin90] Margit Kinder. *Repräsentation mehrdimensionaler funktionaler Abhängigkeiten in neuronalen Netzen.* PhD thesis, Technische Universität München, 1990.

[Kup91] M. Kuperstein. INFANT Neural Controller for Adaptive Sensory-Motor Coordination. *Neural Networks*, 4(2):131–145, 1991.

[LP80] T. Lozano-Perez. Spatial Planning: A Configuration Space Approach. Technical Report A.I. Memo No. 605, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA 02139, 1980.

[Män88] M. Mäntylä. *An Introduction to Solid Modeling.* Principles of Computer Science Series. Computer Science Press, Rockville, ML 20850, USA, 1988.

[May88] Herwig Mayr. A Comparative Survey of Graphic Offline Robot Simulation Systems. Technical Report no. 88-20.1, RISC-Linz, University of Linz, Austria, 1988.

[May91] Herwig Mayr. *Real-Time Dynamic Collision Checking Integrated into Graphic Manufacturing Simulation.* PhD thesis, RISC-Linz, University of Linz, Austria, 1991.

[MH90] Herwig Mayr and Johann Heinzelreiter. The Layer Tree—A Solid Modeling Scheme for Workpieces and Tools. Technical Report no. 90-14.0, RISC-Linz, University of Linz, Austria, 1990.

[MH91] Herwig Mayr and Johann Heinzelreiter. A Solid Modeler for Dynamic Objects Using the Dexel Representation. In V. Hubka, editor, *Proc. International Conference on Engineering Design*, pages 1082–1085. Zürich, Switzerland, August27–29, 1991, Edition HEURISTA, 1991.

[MHÖ89] Herwig Mayr, Martin Held, and Hermann Öllinger. SMART. A Universal System for the Simulation of Machining and Robot Tasks. In *Proc. Computer Applications in Production and Engineering (CAPE '89)*, pages 809–816, Tokyo, Japan, 1989. North Holland, 1989.

[MKSS88] Hiroyuki Miyamoto, Mitsuo Kawato, Tohru Setyama, and Ryoji Suzuki. Feedback-Error Learning Neural Network for Trajectory Control of a Robotic Manipulator. *Neural Networks*, 1(3):251–266, 1988.

[Neu88] NeuralWare, Inc. *NeuralWorks, Revision 2.00*, 1988.

[Pao89] Yoh-Han Pao. *Adaptive Pattern Recognition and Neural Networks.* Addison-Wesley, Reading, Massachusetts, 1989.

[Pau81] R. P. Paul. *Robot Manipulators: Mathematics, Programming, and Control.* MIT Press Series in Artificial Intelligence. Massachusetts Institute of Technology, 1981.

[Pfa93] Jochen Pfalzgraf. On Mathematical Modeling in Robotics. In Jacques Calmet and John A. Campbell, editors, *Proceedings of the Conference on Artificial Intelligence and Symbolic Mathematical Computing*, Lecture Notes in Computer Science, Karlsruhe, Germany, August 1992, 1993. Springer.

[PH87] Cary N. Prague and James E. Hammitt. *Programmieren mit dBase III PLUS.* Markt & Technik, Haar bei München, 1987.

[RHW86] D. E. Rumelhart, Geoffrey E. Hinton, and J. L. Williams. Learning Internal Representations by Error Propagation. In David E. Rumelhart and James L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition; Vol. 1: Foundations*, chapter 8, pages 318–362. MIT Press, Cambridge, 1986.

[RMS89a] Helge J. Ritter, Thomas M. Martinetz, and Klaus J. Schulten. Ein Gehirn für Roboter. *mc*, 89(2):48–61, 1989.

[RMS89b] Helge J. Ritter, Thomas M. Martinetz, and Klaus J. Schulten. Topology-Conserving Maps for Learning Visuo-Motor-Coordination. *Neural Networks*, 2(3):159–168, 1989.

[Roj93] Raúl Rojas. *Theorie der neuronalen Netze.* Springer, Berlin, Heidelberg, 1993.

[Ros58] Frank Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Reviews*, 65:386–408, 1958.

[SHG90] Eberhard Schöneburg, Nikolaus Hansen, and Andreas Gawelczyk. *Neuronale Netzwerke; Einführung, Überblick und Anwendungsmöglichkeiten.* Markt & Technik, Haar bei München, 1990.

[Sim90] Patrick K. Simpson. *Artificial Neural Systems.* Neural Networks: Research and Applications. Pergamon Press, New York, 1990.

[Sixt] Johannes Sixt. Inverse Kinematics in Terms of Denavit-Hartenberg Parameters. Technical report, RISC-Linz, University of Linz, Austria, to appear.

[Was87] Thomas Waschulzik. Optische Mustererkennung in neuronalen Architekturen. Diploma thesis, Technische Universität München, 1987.

[WH60] Bernard Widrow and M. E. Hoff. Adaptive Switching Circuits. In *WESCON Convention Record, Part 4*, pages 96–104. 1960.

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Texts & Monographs in Computer Science. Springer-Verlag, Berlin, third, corrected edition, 1985.

[Wlo92] D. Wloka. *Robotersysteme, Volumes I, II, III.* Springer Verlag, 1992.

[WS63] Bernard Widrow and Fred W. Smith. *Pattern-Recognizing Control Systems.* Computer and Information Sciences Symposium Proceedings. Spartan Books, Washington DC, 1963.

[YG89] Dit-Yan Yeung and George A. Gekey. Using a Context-Sensitive Learning Network for Robot Arm Control. In Hollerbach [Hol89], pages 1441–1447.