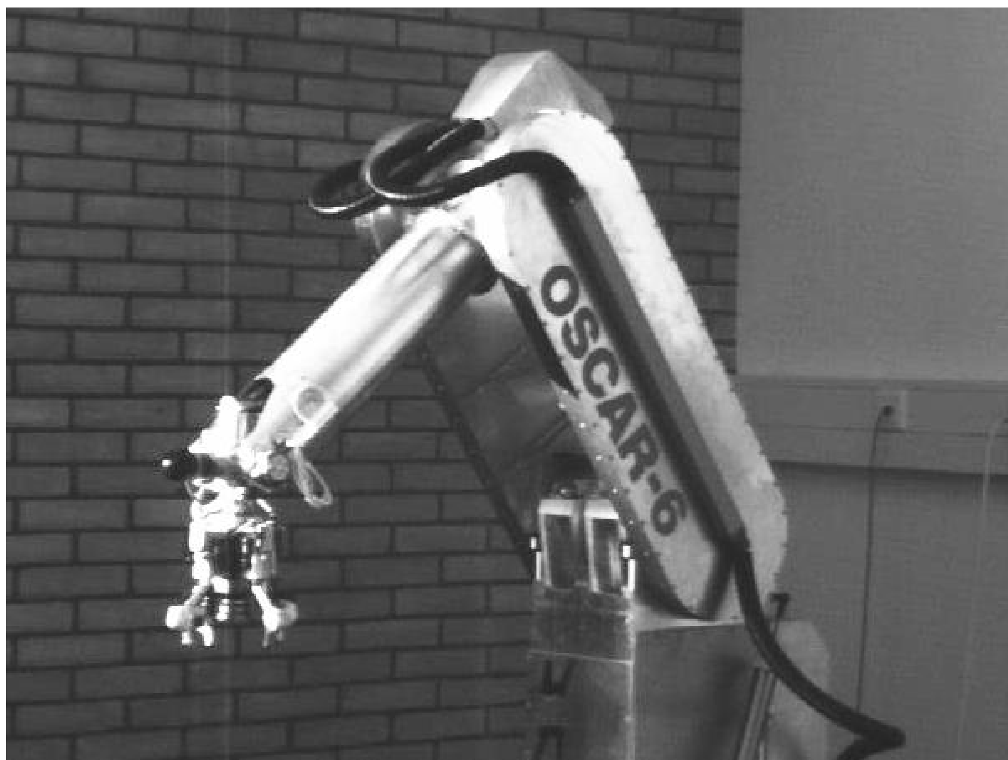
Neural approaches in the approximation of eye-hand mapping and the inverse kinematics function: A Comparative Study

Ferry van het Groenewoud

March 31, 1995



University of Amsterdam
Faculty of Computer Science

Contents

1 Introduction **1**
1.1 Tackling the Problem 1

2 System Description **3**
2.1 The Hardware 3
2.1.1 The OSCAR-6 Robot 3
2.1.2 The camera and end-effector 3
2.1.3 Sun Workstations 4
2.2 The Software 5
2.2.1 The Control Loop 5
2.2.2 Creation of learning patterns 7
2.2.3 Pre-learning 9
2.2.4 Quantizing the error 9
2.2.5 Adaptivity of the system 9
2.3 Non-nested approximators 10
2.4 Nested Approximators 10
2.4.1 The tree concept 10
2.4.2 Propagating learning patterns 11
2.4.3 Split and merge algorithm 11
2.4.4 Bins 13
2.4.5 Adding pre-knowledge 14
2.4.6 Issues on boundaries of the camera inputs 15

3 Neural Network Approaches **16**
3.1 Non-nested approaches 16
3.1.1 Feed-forward networks 16
3.1.2 Kohonen networks 16
3.2 The Nested Network Method 17
3.2.1 Attaching bins to the tree 17
3.2.2 Training of approximators 19
3.2.3 Results 19

4 The Nested Perceptron Approach **20**
4.1 Perceptrons 20
4.1.1 General least squares 20
4.1.2 Incremental property 22
4.2 Attaching bins to the tree 22
4.2.1 Using multiple perceptrons for one approximator 24

4.3 Avoiding the Input Adjustment Method . . . 26
4.3.1 Modifications concerning learning patterns . . . 26
4.3.2 Partially implicit subspace assignment . . . 28
4.3.3 Modifications concerning branch numbers . . . 28
4.3.4 Consequence on performance . . . 29
4.4 Problems with the split and merge algorithm . . . 29
4.4.1 Behavior of the split algorithm . . . 30
4.4.2 Merging nodes wrongly . . . 30
4.4.3 Limitations of the absolute merge algorithm . . . 30
4.4.4 Limitations of the relative merge algorithm . . . 31
4.4.5 Another wrong way to implement merge . . . 33
4.4.6 An alternative way to implement merge . . . 33

5 Performance of Nested approaches 35
5.1 Modifications to the Nested Network approach . . . 35
5.2 Parameters influencing system behavior . . . 36
5.3 Performance of linear interpolation . . . 36
5.3.1 Results using the input adjustment method . . . 36
5.3.2 Results without using the input adjustment method . . . 48

6 Conclusion 58

A Internals of the Software 60
A.1 The data files . . . 60
A.2 The configuration file . . . 61
A.3 Data structures . . . 65
A.4 Complexity and internal mechanisms . . . 67

List of Figures

2.1 Wired frame model of the OSCAR-6 robot 4
2.2 The reach space of the OSCAR-6 robot 5
2.3 An image frame produced by the camera in the end-effector 6
2.4 Control scheme of the robot 8
2.5 Creating multiple learning patterns 9
2.6 Subdividing a two-dimensional space 12

3.1 Feed-forward networks containing 25 and 45 hidden units 17
3.2 Kohonen network; Results of a $7 \times 7 \times 7 \times 7 \times 7$ network 18
3.3 A binary tree with virtual nodes and bins 19

4.1 Bin-sinking; moving bins after a split 24
4.2 Moving bins after a merge 25
4.3 Example of multiple perceptrons linked to a node 27

5.1 Results of learning $\mathcal{F}$ using the input adjustment method with split error $1 \cdot 10^{-4}$. 39
5.2 Results of learning $\mathcal{F}$ using the input adjustment method with split error $1 \cdot 10^{-5}$. 40
5.3 Results of learning $\mathcal{F}$ using the input adjustment method; trials 0-3000 41
5.4 Results of learning $\mathcal{F}$ using the input adjustment method and low split error; trials 0-3000 42
5.5 Results of learning $\mathcal{F}$ using the input adjustment method; Camera rotation of 90 degrees. 43
5.6 Results of learning $\mathcal{F}$ using the input adjustment method; Camera rotation of 30 degrees. 44
5.7 Results of learning $\mathcal{F}$ using the input adjustment method; Elongation of link 1 with 10 centimeters 45
5.8 Results of learning $\mathcal{F}$ using the input adjustment method; Elongation of link 2 with 10 centimeters 46
5.9 Results of learning $\mathcal{F}$ using the input adjustment method; Elongation of link 2 with -10 centimeters 47
5.10 Results of learning $\mathcal{F}$ without using the input adjustment method with split error $1 \cdot 10^{-4}$ 49
5.11 Results of learning $\mathcal{F}$ without using the input adjustment method with split error $1 \cdot 10^{-5}$ 50
5.12 Results of learning $\mathcal{F}$ without using the input adjustment method; trials 0-3000 . . 51
5.13 Results of learning $\mathcal{F}$ without using the input adjustment method and low split error; trials 0-3000 52
5.14 Results of learning $\mathcal{F}$ without using the input adjustment method; Camera rotation of 90 degrees. 53

5.15 Results of learning $\mathcal{F}$ without using the input adjustment method; Camera rotation of 30 degrees. . . . 54
5.16 Results of learning $\mathcal{F}$ without using the input adjustment method; Elongation of link 1 with 10 centimeters 55
5.17 Results of learning $\mathcal{F}$ without using the input adjustment method; Elongation of link 2 with 10 centimeters 56
5.18 Results of learning $\mathcal{F}$ without using the input adjustment method; Elongation of link 2 with -10 centimeters 57

A.1 Internal representation of a tree-node 66
A.2 Internal representation of a pattern-list 67

Acknowledgements

This thesis could not have been written without the help, support and faith of many people. First of all, I would like to thank Patrick van der Smagt for his assistence, suggestions and patience. His brilliant views on the subject matter, his programming skills and remarkable cleverness have impressed me a lot. Besides that, he also is a nice person to deal with and is equipped a great yet subtle sense of humor. It was a pleasure to have him as my supervisor.

Many helpful suggestions were given by Prof. F.C.A. Groen and Dr. B. Kröse, who among other things invented the methodology to solve the problem at hand. Special thanks go to my parents, who motivated me during the entire track, and to Barbara, my girlfriend, who helped me in many ways. They sacrificed much of their spare time to make life easier for me. I also appreciate their interest in my thesis, especially because it takes considerable effort to read without background knowledge on the subjects of artificial intelligence and computer programming. Without their invaluable support in countless ways, I could never have had this work done. I am deeply indebted to them.

Ferry van het Groenewoud, Amsterdam, March 31 1995.

Chapter 1

Introduction

Kinematics deals with the science of motion. This science studies the motion without regard to the forces which cause it. For instance, a study of the position, velocity and acceleration of an object belongs to the domain of the kinematics. A robot consists among other things of a set of joints and a set of links. The joint angles and the length of the links determine the position of the end-effector. When the lengths of the links and the joint angles are known, the position of the end-effector can be calculated. This is called the *forward kinematics function*. The problem discussed in this thesis deals with the reverse problem. Given a position in the world space, a set of joint angles is wanted that will make the robot's end-effector reach this position. This is called the *inverse kinematics function*.

In order to be able to solve equations of the inverse kinematics function the entire geometric description of the robot has to be known in advance. Only when a very accurate description of the geometrical properties of the robot is known, a target position in the world space can be reached with high precision using the inverse kinematics function. Although this solution can be sufficient in many cases, it is very inflexible. As soon as a link of the robot becomes deformed (for instance after a collision) operator intervention is needed to adapt the equations or repair the robot.

Another property of the systems that are investigated in this thesis is that they are equipped with vision. This adds to the complexity of the problem since the position of the target is initially not known in world coordinates, but has to be determined using data that is retrieved from a camera. The system not only has to learn how to move the end-effector from one point to another, it also has to learn how the feedback of the camera should be interpreted. Hence the problem can be described more accurately as eye-hand mapping, with the camera representing the eye, the end-effector representing the hand and the mapping representing a system which handles the eye-hand coordination.

1.1 Tackling the Problem

The method used to solve the eye-hand mapping discussed in this thesis is such that it learns to determine the position of a target object in the world space and reach it without any knowledge of the physical properties of the robot and with limited knowledge of the properties of the camera beforehand. Also, the system is designed in such a way that it is able to adapt to changes that may occur regarding the geometric description of the robot or in the representation of the world space by the camera. This means that the system can adapt to exterior changes without operator intervention. It is important that such adaption be fast, in order to make the resultant system of interest for real-world applicability.

In order to be able to accomplish such a system, the robot has to receive feedback about its state and actions. To fulfill this demand the robot is equipped with joint sensors. The accuracy of the displacements towards the target object can be determined by combining the information retrieved from the camera, which reveals the distance between the end-effector and the target object, and the joint sensors which are used to determine the position of the end-effector. The goal of the system is to reach the target object with the end-effector at a user-defined precision. Specifically, the robot arm must be moved so as to get the target object in the center of the camera image, having a predefined size as observed by the camera. The information that is retrieved by the camera and the joint sensors is used to give the system feedback about how well its attempts to reach the target object are proceeding. Furthermore, correct input-output pairs called *learning patterns* are created with this information, which provides the system with points in the high dimensional function it attempts to approximate.

In this thesis several methods to approximate the eye-hand mapping function are discussed and compared with each other. Earlier results as investigated and described by Arjen Jansen [?], using the Nested Network approach as well as other neural approaches are shortly discussed and placed in perspective with results achieved with the Nested Perceptron approach which has been investigated by us. Also, a look at the implementational aspects will be taken.

In chapter **??** the hardware setup as well as aspects of the software of the system will be described. Earlier attempts to solve the eye-hand mapping in an adaptive way used several kinds of neural networks: feed-forward networks with hidden units, Kohonen networks and the Nested Network method. The latter method makes use of feed-forward networks with hidden units arranged in an hierarchical structure. The functionality and some of the results will be discussed in chapter **??**. The internals of perceptrons as well as the working of the Nested Perceptron approach itself will be discussed in chapter **??**. Just like the Nested Network approach it arranges networks in a hierarchical structure. It differs in the fact that it uses perceptrons instead of feed-forward networks to approximate the eye-hand mapping. A comparison between the several approaches investigated by Arjen Jansen and the Nested Perceptron approach will be made in chapter **??**. An in-depth discussion of the software used to investigate the Nested Perceptron method and the Nested Network method will be given in appendix **??**.

Chapter 2

System Description

2.1 The Hardware

The hardware used to investigate several methods for solving the eye-hand mapping function consists of the OSCAR-6 robot and a video camera. The hardware used to obtain simulation results consisted of various Sun Workstations.

2.1.1 The OSCAR-6 Robot

The robot used for field-testing the system is the OSCAR-6 robot. During development a simulator has been used [?] which simulates a robot which is geometrically equal to OSCAR-6. The OSCAR-6 robot is a six degrees of freedom (DoF) Puma-like robot (see figure ??). Its exact geometric properties are described in [?, Section 2.1]. The system described here is restricted to three DoF, which is sufficient for a wide area of applications like pick-and-place operations.

Figure ?? defines which joint numbers and link numbers are assigned to the joints and links of OSCAR-6. As one would expect from a six DoF robot, it is equipped with six joints. Note that joint i rotates around axis z_{i-1}. The joint angle of some joint i will be denoted as θ_i. For the assignment of joint numbers see also figure ??.

The reach space of OSCAR-6 is depicted in figure ??. The shape of the reach an opened half cylinder with a total volume of 533.6 cubic decimeters. It is bounded according to the following parameters:

$$\begin{array}{rcccl} 46 & \leq & \sqrt{x^2+y^2} & \leq & 93 \\ 0 & \leq & x & \leq & 93 \\ -93 & \leq & y & \leq & 93 \\ 0 & \leq & z & \leq & 52 \end{array}$$

with x, y and z relative to the (x^0, y^0, z^0) coordinate system and expressed in centimeters.

2.1.2 The camera and end-effector

The robot is ego-centered, which means that the camera is placed in the end-effector. Since the choice has been made to build a system that solves the eye-hand mapping for a three DoF robot, constraints have to be made. These consist of:

- The value of joint 4 must always be zero to accomplish the choice that the end-effector has to be perpendicular to the base plane ($z^0 = 0$).

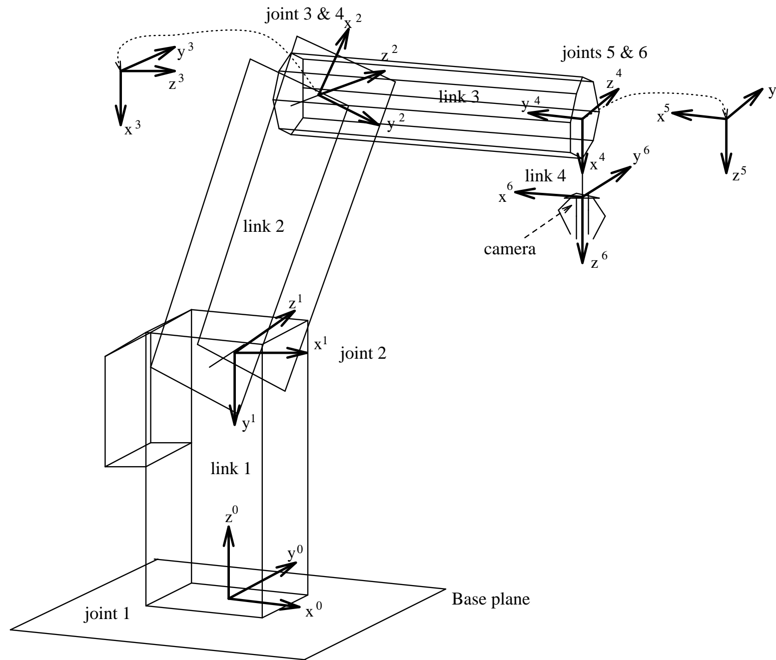


Figure 2.1: Wired frame model of the OSCAR-6 robot. Joint i rotates around axis z_{i-1}. From: [?].

- The value of joint 6 has to be constant and is chosen to be zero.
- To accomplish the choice of the end-effector to be perpendicular to the base plane ($z^0 = 0$) the value of joint 5 must always be equal to $180° - \theta_3$

This causes the end-effector to have an orientation which is such that the camera in the end-effector looks downwards at an angle perpendicular to the base plane: $z^6 \perp (x^0, y^0)$.

The camera will produce an image frame in which the target object is assumed to be situated. A typical image frame is shown in figure **??**. The information retrieved from the image frame will be denoted by $\vec{c}$, which consists of three components:

- The x retrieved from the camera, which is the distance between the x-coordinate of the target position and the center of the image frame,
- In similar fashion the y retrieved from the camera,
- The z retrieved from the camera, which is a measure for the height difference between the end-effector and the target position. The z value from the camera is obtained by measuring the area of the object in the image frame. In the simulated situation however this value is provided directly from the robot simulator software. See also figure **??**.

2.1.3 Sun Workstations

Several types of Sun workstations running SunOS and Solaris under a Unix environment have been used. Here follows a table which shows the performance of these machines. For reference purposes, the speed of a Compaq ProXL Pentium running at 66 Mhz has also been included. This

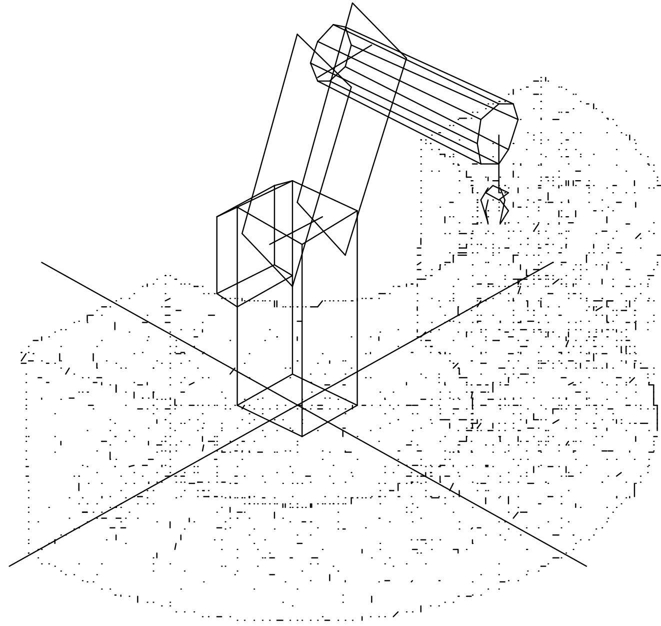


Figure 2.2: The reach space of the OSCAR-6 robot. From: [?].

is useful for making estimates on the run-time of various experiments on machines with a different performance.

Machine	SPECint'92	SPECfp'92
Sun SPARCstation 2	21.8	22.8
Sun SPARCstation 5/70	57.0	47.3
Sun SPARCstation 10/31	45.2	54.0
Compaq ProXL Pentium66	65.1	63.6

2.2 The Software

The software of the system among others controls the behavior of the robot. It receives feedback on what is happening to the robot through camera image frames and joint sensors, and attempts to guide the robot to a randomly chosen target position that resides within the reach space of the robot.

On startup, the system will read a configuration file. This file contains several parameters of the system that can be altered and experimented with by the user. It has been implemented to avoid having to recompile the entire system when parameters are altered. The reader will often encounter the word 'user-definable', which means that the particular parameter that is being described has an entry in the configuration file. In section **??** a typical configuration file has been printed, together with an explanation of all its entries.

2.2.1 The Control Loop

The goal of the system is to move the robot as close as possible to the target position within as few steps as possible. In order to train the system it enters a loop: the Control Loop. This loop is

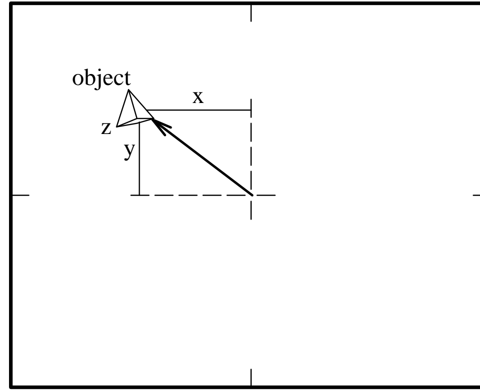


Figure 2.3: An image frame produced by the camera in the end-effector. The object is positioned at the marked spot. The x and y values are directly measured. The z value, a measure for the height difference between the object and the end-effector, is determined by measuring the area of the object.

the same for every method used to solve the eye-hand mapping function discussed in this thesis. One completion of the Control Loop will be called a *trial*. In a trial the system attempts to guide the robot to the target position. The representation of the eye-hand mapping function which the system builds will never be perfect because of the complexity of the function, although the approximation of the eye-hand mapping function gradually improves during the learning process. Hence we will denote the eye-hand mapping function as $\mathcal{F}$ and the system's representation of the eye-hand mapping function as $\mathcal{N}$.

In order to approximate $\mathcal{F}$ as adequately as possible the system builds and trains its internal representation $\mathcal{N}$ using learning patterns. The quality of $\mathcal{N}$ depends solely on the utilisation of learning patterns. The smarter learning patterns are utilized, the better $\mathcal{N}$ will be. $\mathcal{N}$ can be described as being an *approximator* of $\mathcal{F}$. Internally, the approximator $\mathcal{N}$ could consist of one approximator, a lattice of approximators or multiple nested approximators. Methods that use multiple approximators subdivide the entire input space U and assign approximators to subspaces of U. Different approaches on subdividing the input space and utilising learning patterns will be discussed in the following sections.

The robot will virtually never reach the target position within the desired precision in a single feedback step, no matter for how long the system has been trained. This is due to the complexity of $\mathcal{F}$. Therefore multiple steps are generated. During a trial the target does not move. Typically, a maximum of ten feedback steps are generated, with a minimum of three. No further steps are generated if the end-effector has approached the target position within a user-defined distance which will be denoted as ε. In other words, ε is a threshold value which is equal to the desired precision, i.e., the desired maximal distance between the end-effector and the target position. Throughout this thesis, ε can be assumed to be $\frac{1}{2}$ millimeter. This parameter, as well as the number of feedback steps the system should generate are user-definable, but the values just mentioned are most commonly used in this thesis. In case different values were used in an experiment the reader will be notified.

Informally, we refer to s as a step towards the target position, even though a step in some cases increases the distance between the target position and the end-effector. In the Control Loop the following basically happens while taking steps during a trial:

1. A random target position $\vec{p}_{\text{target}}$ is determined within the reach space of the end-effector,

2. The end-effector is moved to an initial starting position $\vec{p}_{\text{endeff}_{s=0}}$,

3. The system generates a step s: $\Delta\vec{\theta}_{s+1} = \mathcal{N}(\vec{c}, \theta_2, \theta_3)_s$ having $\vec{p}_{\text{target}}$ as goal position, the end-effector moves accordingly to a new position $\vec{p}_{\text{endeff}_{s+1}}$,

4. If $\| \vec{p}_{\text{endeff}_s} - \vec{p}_{\text{target}} \| \leq \varepsilon$ or if $s \geq \max(s)$ where $\max(s)$ denotes the maximum number of $\Delta\vec{\theta}$ that are allowed to be generated, this loop is exited.

5. $s \leftarrow s + 1$. Back to 3.

Although the system controls three joints of the robot, it needs only two of them as input for $\mathcal{N}$. So although $\vec{\theta}_{s+1}$ above is a three dimensional vector consisting of θ_1, θ_2 and θ_3, only θ_2 and θ_3 are needed for $\mathcal{N}$. This is also true for $\mathcal{F}$. As a result, $\mathcal{N}$ and $\mathcal{F}$ have a five dimensional vector as their input and a three dimensional vector as their output. Let us make the following additional notational conventions:

- An input pattern will be denoted as $\vec{i} = (\vec{c}, \theta_2, \theta_3)$,
- An output pattern will be denoted as $\vec{o} = \mathcal{N}(\vec{i}) = (\Delta\theta_1, \Delta\theta_2, \Delta\theta_3)$ i.e., the delta joint values generated using the internal representation of $\mathcal{F}$ of the system,
- A target pattern will be denoted as $\vec{t} = \mathcal{F}(\vec{i}) = (\Delta\theta_1, \Delta\theta_2, \Delta\theta_3)$, i.e., the desired delta joint values that bring the end-effector exactly to the target position,
- A learning pattern will be denoted as $\vec{l} = (\vec{i}, \vec{t})$.

The goal of the system can now be denoted as to minimize:

$$\sum_p \|\mathcal{N}(\vec{i}_p) - \mathcal{F}(\vec{i}_p)\|$$

2.2.2 Creation of learning patterns

In order to be able to let the robot move towards the target and ultimately approach the target within the desired precision ε, it needs information about the target position and its own current position. Using this information, it generates a step s that ideally should move the robot closer towards the target position. This however does not always happens as will be discussed later.

The two dimensional vector $\vec{\theta}$ that is fed to $\mathcal{N}$ consists of θ_2 and θ_3 as noted earlier. It can be shown that the current state of the base rotation θ_1 is not needed, since the camera is attached to the end-effector and is rotated just as much has the robot itself, and since only $\Delta\vec{\theta}$ values as opposed to absolute joint values are used to guide the robot towards the target position. The amount of base rotation required is solely dependent of the x and y values retrieved from the image frame and independent of the current rotation of the base θ_1. For a mathematical prove refer to [?, figure 2.2].

Input vector $\vec{i}$ defines the state of the robot and the target position. All information is present to be able to calculate an output vector $\vec{o}$ needed to move the robot exactly to the target position, provided that the Denavit-Hartenberg description of the robot is known to the system or $\mathcal{N} \equiv \mathcal{F}$. Such is not the case. The system has to learn how to move to the target position using the information it receives during the learning process. While this process is progressing the approximation of $\mathcal{F}$, i.e., $\mathcal{N}$ generally improves.

During the learning process the system generates a large number of learning patterns $\vec{l} = (\vec{i}, \vec{t})$. The key to the learning process is the ability of the system to create such learning patterns, which

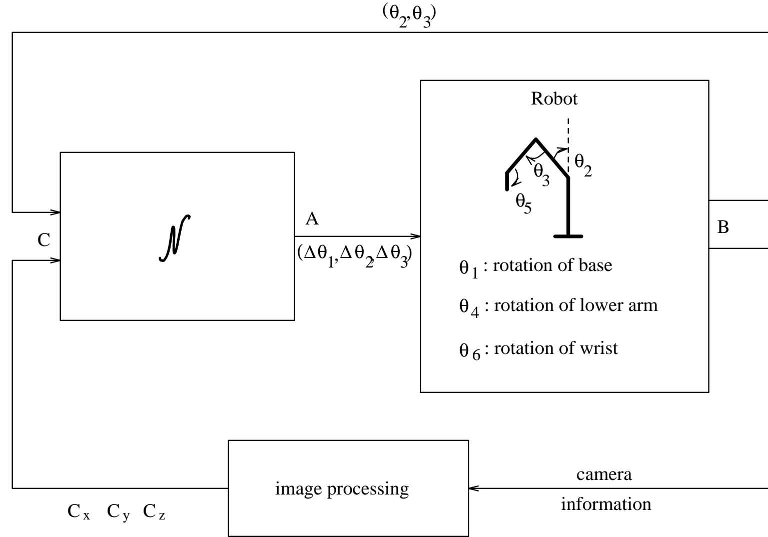


Figure 2.4: Control scheme of the robot. From the current robot position determined by θ_2 and θ_3, and the target position determined by the camera image a unique input vector $\vec{i}$ coming from the robot at point B, can be fed to the controller at point C. The controller has to generate from this input vector a robot displacement $\Delta\theta_1$, $\Delta\theta_2$ and $\Delta\theta_3$ (point A). This displacement has to bring the object in the center of the camera image at a certain size. This process can be repeated if the previous displacement is not accurate enough and are called the feedback steps of the system. From: [?].

means that the exact delta joint values needed to reach a position in the world space, i.e., $\vec{t}$ have to be known.

When at step s an input pattern $\vec{i}_s$ is generated, the system calculates an output pattern $\vec{o}_s = \mathcal{N}(\vec{i}_s)$. $\vec{o}_s$ is sent to the robot which moves its end-effector accordingly. Ideally it places its end-effector within distance ε from the desired target position, however this does not happen in most cases, especially when the system is in an early stage of the learning process. The system then retrieves once again information from the camera to calculate its current position and calculates what the desired movement would have been, i.e., $\vec{t}_s$ using the *input adjustment method.* Consequently the system constructs a learning pattern $\vec{l} = (\vec{i}, \vec{t})$ which is a point in $\mathcal{F}$. The more points of $\mathcal{F}$ are known to the system, the better its approximation $\mathcal{N}$ will become. For a discussion of the input adjustment method refer to [?, Appendix C].

If the system takes s steps towards a target position during a trial, it is possible to create

$$1 + 2 + \cdots + s = \frac{s(s-1)}{2}$$

learning patterns for each trial. See figure **??** for a clarification.

To give a quick indication of the number of learning patterns the system creates: when the number of steps in one trial is 10, the number of learning patterns may reach 8,000 within 500 trials as can be seen from figure [reference].

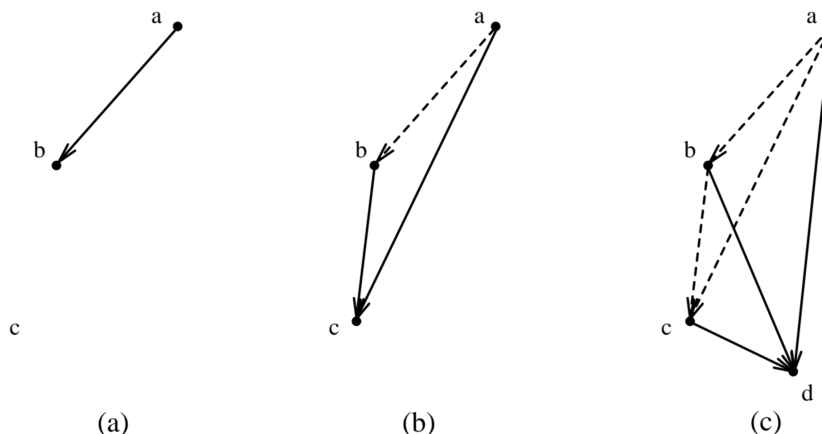


Figure 2.5: Creating multiple learning patterns from a particular end-effector position. The end-effector of the robot travels path a, b, c, d (three steps). After the movement from a to b, a learning pattern can be calculated from this movement. If the end-effector continues and reaches c (b), not only can a learning pattern be calculated using the movement from b to c. From the information of the positions of a and c yet another learning pattern can be calculated (b). If the end-effector continues and reaches d, three additional learning patterns can be calculated, shown with solid lines in (c).

2.2.3 Pre-learning

The system starts learning without any knowledge of $\mathcal{F}$. In order to give the system a minimal amount of knowledge about the function it has to approximate, pre-learning is used. One pre-programmed target position is generated, as well as a number of pre-programmed robot moves. These moves generate learning patterns which are learned by the system. The learning patterns are chosen such that it prevents the system from sticking to only a small portion of the total reach space of the robot. After pre-learning, it will be almost certain that the entire reach space will be explored. The number of learning patterns generated with pre-learning is very small compared to the number of total learning patterns that will be generated when the system starts learning with randomly generated target positions. Usually, 8 pre-learning moves are done.

2.2.4 Quantizing the error

The error of n learning patterns is determined by calculating the average *sum-squared error* of those learning patterns. The sum-squared error of a single learning pattern is calculated as follows. The input vector $\vec{i}$ yields an output value $\vec{o} = \mathcal{N}(\vec{i})$. The desired output value $\vec{t}$ then is calculated using the input adjustment method. The error E of a single learning pattern $\vec{l}$ equals:

$$E_{\vec{l}} = \| \vec{o} - \vec{t} \|$$

In words, this is the distance measured in degrees by which the joints miss the target position.

2.2.5 Adaptivity of the system

A system is wanted that not only is able to guide the robot to the target, but which is also able to quickly adapt when changes on the robot or camera occur. For instance, the camera could be rotated, or one of the links of the robot could be damaged and therefore be deformed. This

would result in a different $\mathcal{F}$. The system should behave such that it can adapt to an altered $\mathcal{F}$ quickly. Although in early approaches some facilities were already implemented to accomplish this, they relied mainly on the adaptive behavior of feed-forward networks and Kohonen networks. The Nested Network and Nested Perceptron methods use a more sophisticated approach which enables the system to adapt much faster to changes caused by external influences.

2.3 Non-nested approximators

Using only one approximator to approximate $\mathcal{F}$ means that the input space has not been divided in any subspaces. All generated learning patterns are used to train a single approximator that represents the entire input space. The results of an instance of this approach are discussed in section **??** in which a feed-forward network with hidden units is used as approximator.

The input space U can be divided into subspaces, and to each subspace an approximator can be assigned. The subdividing of U is done by dividing each dimension of U in a fixed number of equally large subspaces. For instance, a two dimensional space U^2 would be split in four subspaces if each dimension of U^2 would be split in half. Since $\mathcal{F}$ is a five dimensional function, halving each dimension would result in 32 subspaces. An approximator assigned to a subspace then only has to approximate $\frac{1}{32}$ part of the input space, which results in a more accurate approximation of that subspace and consequently of $\mathcal{F}$. Results of this approach are described in section **??**. U was divided in $7^5 = 16,807$ subspaces. It was implemented using a $7 \times 7 \times 7 \times 7 \times 7$ Kohonen network.

2.4 Nested Approximators

The complexity of $\mathcal{F}$ will not be the same at any interval. For instance, $\mathcal{F}$ can be smooth and nearly linear at some subspaces, while being highly non-linear and having many discontinuities at other subspaces. This property can be taken advantage of by dynamically assigning approximators. A subspace of U can be divided if $\mathcal{F}$ appears to be more complex for that part of the input space, while other subspaces of U can remain undivided if the approximation suffices there. The result is a multi-resolution approximation of $\mathcal{F}$. The following description of the internal representation of nested approximators applies to the Nested Network method as well as to the Nested Perceptron method. Because differences in the internal representation exist between these two methods the description below is not complete. In chapters **??** and **??** the method-specific details will be discussed.

2.4.1 The tree concept

In order to be able to dynamically assign approximators, a tree structure is used. The root of the tree holds one approximator that represents the entire input space U. Initially, the root node is the only node that exists. Each node in the tree has 2^I branches and consequently can maximally have 2^I children, with I denoting the dimensionality of the input space ($I = 5$ in our case). A child node $N(b)$ of a node N with b representing a branch is constructed by halving each dimension of subspace U_N and choosing the appropriate half, i.e., choosing each of the I halves such that $N(b)$ represents $U_{N(b)}$. Since there are for each dimension two halves to be chosen from, a binary digit is used to represent either of the halves. Each branch of a node is given a value

$$b = [\beta_{I-1}\beta_{I-2}\dots\beta_0]$$

with $\beta \in \{0,1\}$ and b equaling

$$b = \sum_{j=0}^{I-1} \beta_j 2^j$$

As a result, $0 \leq b < 2^I$. Now, each node in the tree can be uniquely identified by d branch numbers:

$$N = \langle b_1, b_2, \ldots, b_d \rangle$$

where d denotes the depth of the node in the tree. N represents an input space

$$U_N = U_{\langle b_1, b_2, \ldots, b_d \rangle}$$

The root node resides at depth 0 by definition. Note that

$$U_{\langle b_1, b_2, \ldots, b_d \rangle} \subset U_{\langle b_1, b_2, \ldots, b_{d-1} \rangle} \subset \ldots \subset U \tag{2.1}$$

and that the number of elements in the array N defines the depth of N in the tree and subsequently of U_N. A single node N also will be denoted as $N[d]$ with d denoting the number of elements in N. Consequently, $U_{N[d]}$ denotes a single subspace of U at depth d. Note that it does not define which subspace. For an example of subdividing a two-dimensional input space U^2 refer to figure **??**. Subdividing a subspace $U_{N[d]}$ by adding a node that represents a subspace $U_{N[d+1]} \subset U_{N[d]}$ will henceforth be called a *split*.

The depth of the tree will never exceed a user-definable maximum, which will be denoted as D; $0 \leq d \leq D$. The input space will be split until for every subspace U_N a desired user-definable precision has been reached (see section **??**). However, a subspace $U_{N[D]}$ will not be split any further even if the desired precision is not achieved for $U_{N[D]}$ since the depth of the tree would then exceed D.

The result of this mechanism is that at lower depths in the tree and particularly in the root node a coarse approximation of $\mathcal{F}$ exists, while at higher depths a fine approximation of $\mathcal{F}$ exists.

2.4.2 Propagating learning patterns

A learning pattern belonging to a subspace $U_{N[D]}$ will be trained to all existing approximators representing subspaces $U_{N[D]} \subset U_{N[D-1]} \subset \ldots \subset U_{N[0]}$. $N[0]$ represents the root node and consequently $U_{N[0]}$ represents the entire input space U. Note that every new learning pattern will be trained to the root node. The result of this mechanism is that nodes at higher levels in the tree will receive more and thus newer learning patterns than nodes residing at deeper levels in the tree. Consequently, approximators at higher levels in the tree will adapt relatively quickly to a change in $\mathcal{F}$ such that a coarse approximation of $\mathcal{F}$ will be re-established more quickly than a fine approximation. From this it is also clear that any new learning pattern will be learned to the deepest existing approximator that represents the subspace of that learning pattern.

2.4.3 Split and merge algorithm

The decision whether a subspace $U_{N[d]}$ should be split creating a new node $N[d+1]$ representing a particular subspace $U_{N[d+1]} \subset U_{N[d]}$, will be taken regarding three conditions:

- A split should not make the tree deeper than its maximum depth, i.e., $d + 1 \leq D$.
- A minimum number of learning patterns belonging to $U_{N[d+1]}$ is required to exist.
- The produced error of selected learning patterns that belong to $U_{N[d+1]}$ must exceed a user-definable value. This value will hence be called the *split error*.

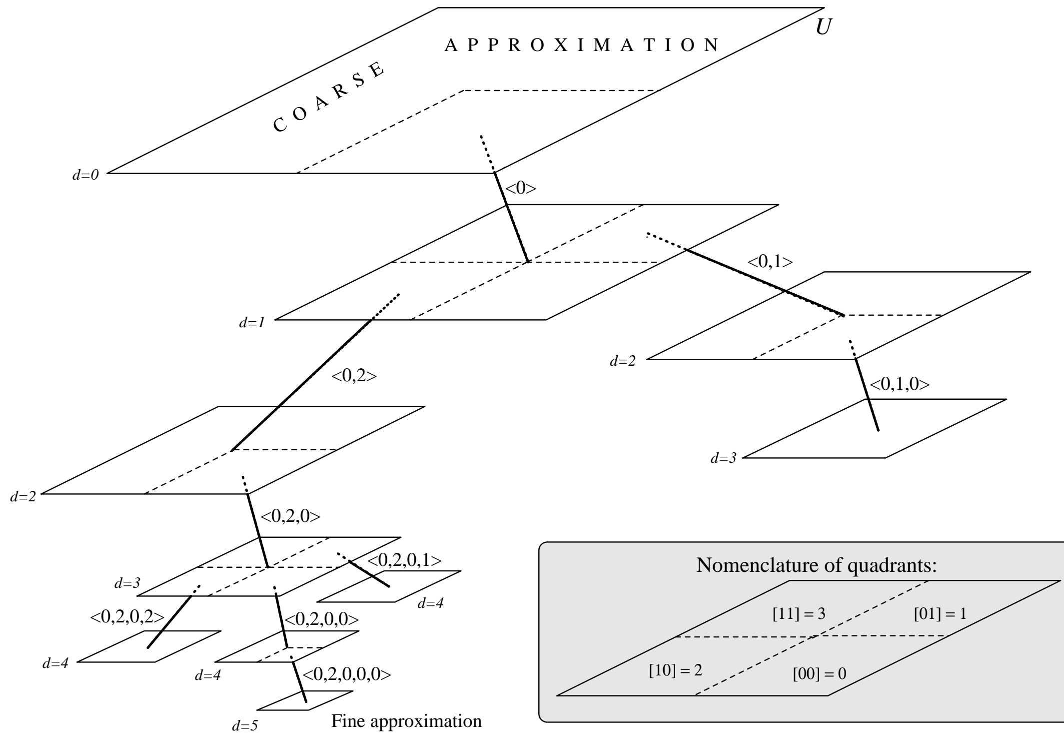


Figure 2.6: Example of the subdividing of a two-dimensional space U^2. Each space can be divided into four equally sized parts. If no further subdividing exists the best possible approximation for that specific subspace is reached. From: [**?**].

If those conditions are fulfilled, a split is executed. The newly created node $N[d+1]$ will be trained immediately with selected learning patterns belonging to $U_{N[d+1]} \subset U_{N[d]}$.

Several measures have been taken to retain adaptivity when minor changes in $\mathcal{F}$ occur and to facilitate the detection of major and sudden changes in $\mathcal{F}$. Major changes are caused by for instance a rotation of the camera by 90 degrees, or by elongating a link considerably. In these cases it is desirable to discard all learning patterns and relearn the new $\mathcal{F}$ from scratch instead of gradually adapting, which will take more trials or can in some cases even be impossible. Therefore, the system is able to cut branches and its nodes $N[d+i]$ representing a subspace $U_{N[d+i]} \subset U_{N[d]}$, i.e., *merge* parts of the tree. Two kinds of merge exist: *absolute merge* and *relative merge*.

An absolute merge is executed when the error in a node becomes higher than a user-definable threshold value. This is done in order to be able to merge a node if its error becomes very large, for instance after a rotation of the camera of 90 degrees. Therefore, each approximator calculates the average error that the most recent n (user-definable) learning patterns have produced. Every time some approximator A receives a new learning pattern, and at least n learning patterns have been learned to A, the average error that the most recent n learning patterns produced is updated. This error is used to decide if a node needs to be merged by comparing it to a user-definable parameter called the *absolute merge error*. The reason that the average error based on the most recent n learning patterns is calculated is because of the fact that the error that an approximator produces shows large fluctuations. By taking the average error of the most recent n patterns these fluctuations can be largely reduced.

The situation in which an absolute merge is required occurs after a major change in $\mathcal{F}$, for instance, after a 90 degrees camera rotation. After such a major change, the system fails to deliver correct learning patterns. A vicious circle is then entered: approximators do not receive correct learning patterns because of the large errors they produce, and because of the incorrect learning patterns, approximators keep producing large errors. That is why, when an approximator shows a large error, it should be merged using absolute merge.

A relative merge is executed if some node $N[d]$ approximates a particular subspace $U_{N[d+1]} \subset U_{N[d]}$ better than $N[d+1]$ by a user-definable threshold value which will henceforth be called the *relative merge error*. When some node $N[d]$ receives a child node $N[d+1]$, $N[d]$ will start measuring how well it approximates $U_{N[d+1]}$. It does this by separately measuring the error that learning patterns belonging to $U_{N[d+1]}$ produce when they are tested in $N[d]$. This error is averaged over a user-definable number of learning patterns, identically to the way this is done with the absolute merge error. The result is compared to the absolute merge error of a child node $N[d+1]$, which represents the error of the same part of the input space. If the absolute merge error of that particular child node is significantly higher than the error that the parent node produces for the same part of the subspace, the child node and all its children will be merged. The user decides what is significant in this case by determining an appropriate value for the relative merge error. Formally, a node $N[d+1]$ such that $U_{N[d+1]} \subset U_{N[d]}$ is merged if

$$E_i\langle b_1, b_2, \ldots, b_d\rangle > E\langle b_1, b_2, \ldots, b_d, i\rangle + \epsilon_r \tag{2.2}$$

where E denotes the average absolute error of a node $N = \langle b_1, b_2, \ldots, b_d\rangle$, E_i denotes the error of node N for learning patterns belonging to the subspace that is also represented by the child node at branch number i, and ϵ_r denotes the relative merge error.

The situation in which nodes at deeper levels approximate a subspace worse than nodes at higher levels arises mainly after a minor change of $\mathcal{F}$. Nodes at higher levels adapt more quickly to a change of $\mathcal{F}$ than nodes at deeper levels, since the former receive more learning patterns. This is caused because of the fact that nodes at higher levels represent a larger part of the input space. Therefore, at some point in time a coarse approximation of a subspace can become significantly better than a fine approximation. At that point, a relative merge is executed.

A merge of a node $N[d]$ will recursively delete all existing nodes $N[d], N[d+1], \ldots, N[D]$ with $U_{N[d]} \supset U_{N[d+1]} \supset \ldots \supset U_{N[D]}$. Simply put, the subtree at $N[d]$ is cut from the tree as well as $N[d]$ itself.

2.4.4 Bins

To be able to train a newly created node with learning patterns created earlier in the learning process and to be able to calculate the error of a node using earlier created learning patterns, learning patterns are stored. Learning patterns belonging to a subspace $U_{N[D]}$ are stored in a *bin* representing just that subspace. A bin can hold a user-definable maximum of learning patterns. If a bin is full while a new learning pattern has to be stored in that bin, the oldest learning pattern will be discarded in favor of the new learning pattern. A bin can be uniquely identified by the array

$$\langle b_1, b_2, \ldots, b_D\rangle$$

where b is a branch number and D is the maximum depth of the tree. Note that a bin represents a smallest subspace of U that can be represented by an approximator in a tree of depth D. Learning patterns stored in an arbitrary bin represent the following subspaces: $U_{N[D]} \subset U_{N[D-1]} \subset \ldots \subset U_{N[0]}$. This is a consequence of equation (**??**).

A quick calculation shows that the maximum number of bins of a five dimensional tree with a maximum depth of 6 amounts to $32^6 = 1{,}073{,}741{,}824$, i.e., more than a billion. It is clear that

memory-wise, this number of bins can never be created. This is not necessary since the number of learning patterns created during the learning process typically reaches a maximum of 20,000 for a run with a reasonable number of trials. The number of learning patterns that are created during learning is always at least the same but generally more than the number of bins since multiple learning patterns can reside in one bin.

To be able to retrieve learning patterns belonging to a certain subspace $U_{N[d]}$ without excessive computational overhead the bins are attached to the tree in such a way that it will not be necessary to visit every bin to see if it holds learning patterns that belong to $U_{N[d]}$. Since the Nested Network Method and the Nested Perceptron Method each use a different method to attach bins to the tree, the details will be described separately in chapters ?? and ??.

2.4.5 Adding pre-knowledge

One of the problems with approaching a target position closely with the end-effector is that once the end-effector is very near, say well within the limit of ε, it can direct the end-effector to a position well out of the ε range the next feedback step. This is caused by the sharp boundaries between the subspaces that different approximators represent. If the target position lies close to a boundary of two or more different approximators, the situation can occur that some approximator A leads the end-effector within range ε towards the target position, but at the same time also within the subspace of another approximator B which then takes over and will generate a movement at the next feedback step. If approximator B is less well suited to approximate that edge of its subspace as approximator A, it will generate a movement well away from the target position. There is a way however to deal with this problem.

The goal of the system is to approach a randomly chosen target position within range ε with the end-effector of the robot. This is translated into attempting to get the target object in the middle of the camera image frame at a specified size, i.e., to have $\vec{c} = (0,0,0)$. As soon as this position has been reached, no further movement is required. This translates into learning patterns that have the form

$$\vec{l} = (\vec{i}, \vec{t}) = (c_x, c_y, c_z, \theta_2, \theta_3, \Delta\theta_1, \Delta\theta_2, \Delta\theta_3) = (0,0,0,\theta_2,\theta_3,0,0,0)$$

In other words: if the distance between the end-effector and the target object is zero, no movement is wanted. These kind of learning patterns will hence be referred to as *known patterns*. Known patterns can be trained to the system at a user-definable percentage. When some node at some point is trained with n regular learning patterns, the system will also train that node with $(p/100)n$ known patterns, where p denotes a user-definable percentage of known patterns that should be added. Known patterns are trained to a node on the fly, directly after regular learning patterns have been trained to it. The variable parts are chosen randomly, but such that they fall into the input space of the node they are trained to. Known patterns are not stored since they can be created at any time they are needed, at virtually no computational cost.

Adding known patterns is not without any side-effects. Setting the percentage of known patterns to be added very high (for instance to 95%) will have the effect that the end-effector hardly moves anymore during a feedback step. In general, it can be noted that adding known patterns enhances the precision of the system, but raises the average number of feedback steps that have to be taken. The benefit of this is that the end-effector tends to stay near the target position as soon as it has approached it closely during a trial, even when the target position is near the edge of multiple approximators.

2.4.6 Issues on boundaries of the camera inputs

Some precaution has to be taken when assigning the boundaries of the camera inputs. Recall that when a node is being split, each dimension is divided exactly in the middle of the current boundaries. Then, the appropriate halve is selected for each dimension. While $\mathcal{N}$ gradually improves, the camera inputs $\vec{c}$ will lie closer to $(0,0,0)$. It represents the part of the input space where the end-effector is close to the target position. One would want the representation of this particular part of the input space to be very precise. Therefore it is undesirable if 0 in any of the camera inputs would lie close to a boundary of an approximator at any depth, since that would require extrapolation instead of interpolation to reach the target. Optimally, $\vec{c} = (0,0,0)$ should lie exactly in the middle of the camera boundaries, regardless the depth of a node, in order to get an as good as possible representation of $\vec{c} = (0,0,0)$. Unfortunately, this is impossible to achieve. Consider the following boundaries of the camera vector $\vec{c} = (c_x, c_y, c_z)$:

$$-a_x < c_x < a_x; \quad -a_y < c_y < a_y; \quad -a_z < c_z < a_z$$

When using these boundaries, zero is exactly in the middle of all three camera inputs for what the root node is concerned. However, when the root node is split, the boundaries of $\vec{c}$ are cut right in the middle, and zero would end up at the edge of all camera inputs, the worst thinkable situation. One way around this situation would be to split the camera dimensions in three equally sized parts instead of two, to keep zero exactly in the middle of one of those parts. Another solution, which has been applied, is to chose the boundaries such that zero can never end up at a boundary. Consider the following boundaries:

$$-2a_x < c_x < a_x; \quad -2a_y < c_y < a_y; \quad -2a_z < c_z < a_z$$

Zero is not in the middle, but at two third between the boundaries. If, for example, the input space of input c_x is divided in two equally sized halves, the boundaries around c_x would become

$$-2a_x < c_x < -\frac{1}{2}a$$

for the left halve, and

$$-\frac{1}{2}a_x < c_x < a_x$$

for the right halve. Notice that zero is positioned at one third of the right halve. Would this boundary be divided in a left and right halve again, zero would end up at two third of the left halve. In other words, when using boundaries such that zero is positioned at either one third or two third between those boundaries, zero will alternate between one third and two third, and would never end up at a boundary.

A consequence of this method is that it places constraints on the camera input boundaries. For instance, if the real camera boundaries are $-a$ and a, then one is forced to use boundaries $-2a$ and a or $-a$ and $2a$ to position zero at one third or two third between the boundaries. However, this will not influence any aspect of performance or memory consumption due to the split and merge algorithm. Since some parts of the input space will receive no learning patterns at all, they will never be split. If one would be concerned about the fact that the camera input dimensions will be relatively large compared to the joint input dimensions due to the enlargement, one could just enlarge the joint input boundaries as well.

Chapter 3

Neural Network Approaches

Previously reported methods for approximating the eye-hand mapping function include feed-forward networks, Kohonen networks and the Nested Network Method. The results will be briefly discussed; for a more detailed discussion on these methods and their results can be found in [?].

3.1 Non-nested approaches

Below follows a description of results achieved with feed-forward and Kohonen networks. These methods have only limited facilities implemented to adapt to changes in $\mathcal{F}$.

3.1.1 Feed-forward networks

Using a feed-forward network trained with conjugate gradient back-propagation containing 25 hidden units the results depicted in figure **??** were achieved. As can be seen from the figure, the distance of the end-effector and the target will average around 4 millimeter after 8 steps towards the target position per trial. Arjen Jansen's figures show that increasing the number of hidden units to 45 does not have any significant positive effects. Using one feed-forward network with hidden units is not a satisfactory solution to approximate $\mathcal{F}$ since a higher precision is desired. A major problem with the network used here is that it can not cope with the large input space and complexity of $\mathcal{F}$.

3.1.2 Kohonen networks

Another approach investigated by [ref to Ritter & Martinetz] make use of Kohonen networks. A detailed discussion of Kohonen networks can be found in [?] and [?]. In short, Kohonen networks consist of a lattice of neurons, forming a topologically correct map of the input space. Each neuron in the lattice represents and approximates a fixed subspace. The simulation results are slightly better than those using the method as described in section **??**. Figure **??** show results achieved using a $7 \times 7 \times 7 \times 7 \times 7$ Kohonen network. It can be seen that the network needs more trials before the point is reached at which the distance between the end-effector and the target position does not improve any further. Although this distance is less than when using the method discussed in section **??**, the maximum precision reached after 2 steps still is about 5 millimeter. This is about the maximum precision that can be attained using this Kohonen network. It took a Sun SPARCstation 1+ about two days to reach 20,000 trials. Although expanding the Kohonen network with more neurons would result in a better approximation of $\mathcal{F}$, it would cause the run-time to become unreasonably long.

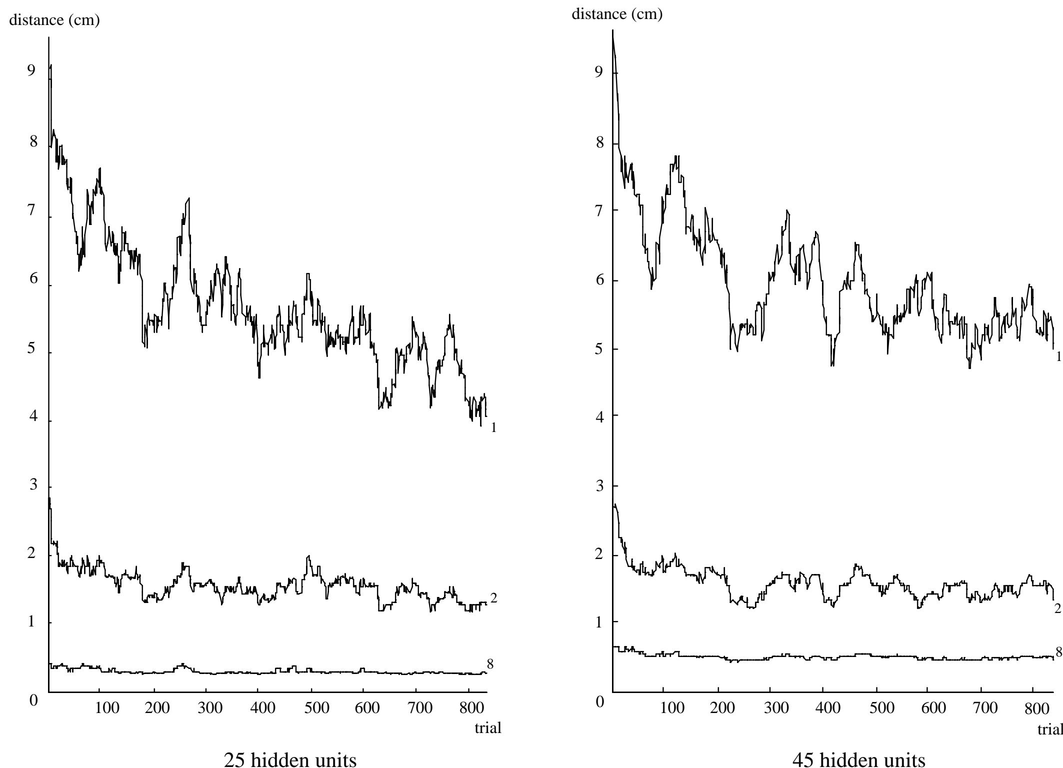


Figure 3.1: Feed-forward networks containing 25 and 45 hidden units. The distance between the end-effector and the target object in cm. are shown after 1, 2 and 8 feedback steps. Using a feed-forward network containing 45 hidden units does not show any significant positive effects. Both plots are smoothed using the average of 50 points. From: [?].

3.2 The Nested Network Method

The use of a single neural network in the previously described methods has many disadvantages. When a feed-forward network with hidden units is used, the network will quickly learn and adapt, but the quality of the approximation of $\mathcal{F}$ is not satisfactory. Using Kohonen networks results in a more precise approximation of $\mathcal{F}$, but it takes longer to train it. Therefore, a different approach has been developed which combines the fast learning performance of the feed-forward networks with subdividing the input space as used in Kohonen networks; the Nested Network Method. It uses a tree structure as was described in section **??**.

3.2.1 Attaching bins to the tree

A bin represents a subspace of $\mathcal{F}$ at the finest possible granularity that can be attained with a tree having maximum depth D. It represents a subspace $U_{N[D]}$ that is also represented by a leaf node $N[D]$. Since leaf nodes also represent subspaces of all its ancestors it is convenient to attach bins to the leaves of the tree. If learning patterns for $U_{N[d]}$ have to be collected, learning patterns residing in bins attached to leaves $N[D]$ such that $U_{N[D]} \subset U_{N[d]}$ are collected recursively. This can be achieved by for example a depth-first travel through the appropriate subtree since all learning patterns that reside in bins that are attached to the leaves of the subtree belong to the

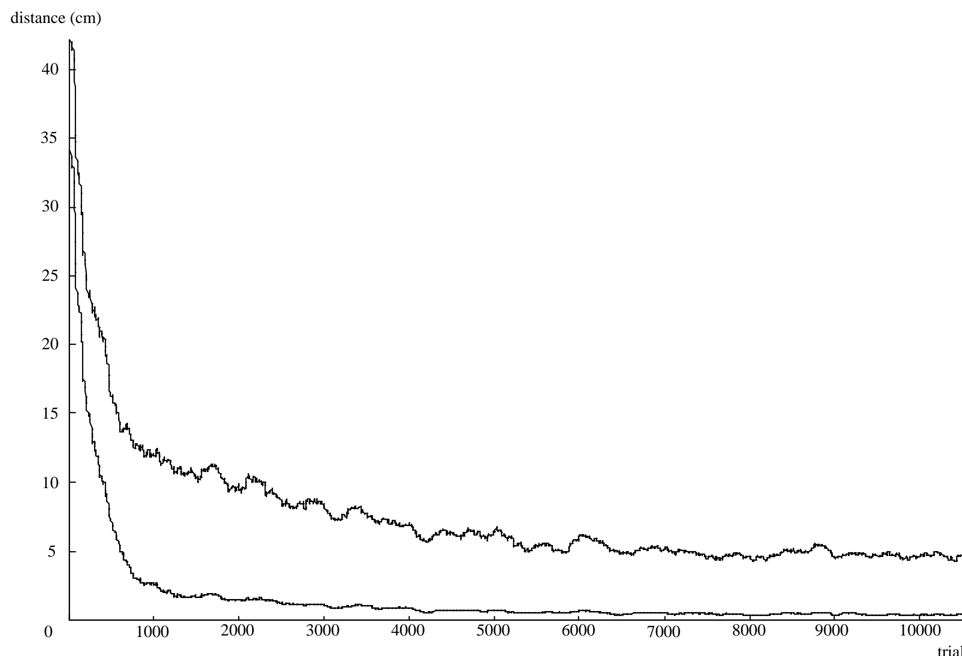


Figure 3.2: Kohonen network; Results of a $7 \times 7 \times 7 \times 7 \times 7$ network. The precision after a gross move (1) and a fine move (2) is displayed. This plot is smoothed using the average of 50 points. From: [?].

subspace the subtree represents. This mechanism enables the system to retrieve learning patterns of a particular subspace without excessive computational overhead.

Since learning patterns are stored in bins at depth D, all nodes representing subspaces

$$\vec{l} \in U_{N[D]} \subset U_{N[D-1]} \subset \ldots \subset U_{N[1]}$$

are created if they do not yet exist. The only function of nodes that are created on such occasion is to provide a path to a bin attached to a leaf node at depth D. These nodes are called *virtual nodes*. Virtual nodes do not contain an approximator, and the creation of a virtual node is not regarded to be a split. However, if at some point in time a virtual node receives an approximator, it will be called a split.

As soon as the first learning pattern is created, a path to the bin in which the learning pattern has to be stored is calculated, and a number of virtual nodes are created, one less than the maximum depth of the tree (since the root node already exists and is not a virtual node). After a while, when many learning patterns are stored, fewer virtual nodes for each learning pattern have to be created on the average because many virtual nodes and bins will already exist. See figure **??** for an example of a binary tree with virtual nodes and bins.

Virtual nodes and bins provide a convenient way to store and retrieve learning patterns from a computational point of view. However, this method has a significant drawback concerning memory requirements. The number of bins required will never exceed the number of learning patterns created during the learning process (see also section **??**). The number of required virtual nodes however will be higher. Experimentally it has been established that 1,500 trials require about 50,000 virtual nodes to be created. Since each virtual node takes up about 500 bytes of memory, roughly 25 megabytes of memory are required to accommodate virtual nodes alone.

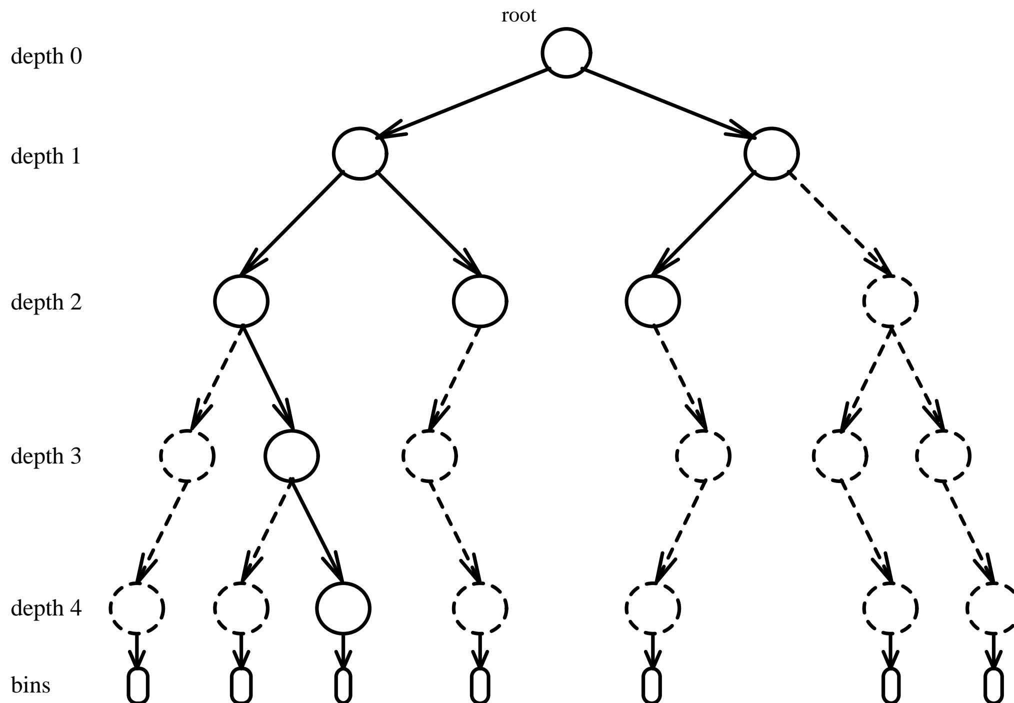


Figure 3.3: An example of a binary tree with virtual nodes and bins. Normal nodes are drawn with solid lines, virtual nodes are drawn with dashed lines. Pointers that point to a virtual node are also drawn with dashed lines.

3.2.2 Training of approximators

Feed-forward networks can be incrementally updated. This means that if a feed-forward network receives a new learning pattern, it does not have to reset all weights of the neurons and relearn its network from scratch. Training a feed-forward network is computationally very expensive. On a Sun SPARCstation 20/51, during training a single feed-forward network with 5 hidden units a speed of only ten learning patterns per second is reached. Therefore, the networks are trained with back-propagation after each step of a trial, which is relatively cheap. After each trial, however, each network that has received a learning pattern will be relearned to regain a more optimal representation of its subspace. It is relearned with the most recently added learning pattern of every bin in its subspace.

3.2.3 Results

Using the Nested Network Approach, a precision of 1 millimeter after three feedback steps can be achieved after about 1,200 trials. The precision for three feedback steps will not improve any further. After 1 feedback step, the distance between the end-effector and the target object is 30 millimeter when 2,000 trials have passed. It creates about 1,800 networks and 8,000 bins within 2,000 trials.

Chapter 4

The Nested Perceptron Approach

In the Nested Perceptron Approach the same split-and-merge algorithm has been used as in the Nested Network Approach. The Nested Perceptron Approach uses a tree-structure as described in **??**. Instead of a feed-forward network with hidden units however, its approximators consist of single-layer *perceptrons*. A detailed description of perceptrons will be given in this chapter. Bins are linked to the tree in a different way; virtual nodes have become obsolete, which reduces memory requirements. These issues will be thoroughly discussed.

4.1 Perceptrons

The disadvantage of using a feed-forward network as approximator is that it is a very expensive method from a computational point of view. The reason for this is that in order to retain an optimal representation for an ever growing set of learning patterns, the approximator is relearned after trials in which new learning patterns were added. Since every learning pattern is also learned to all ancestors of a node, the number of approximators that has to be relearned is considerable when the number of nodes and learning patterns become large (e.g., after 1,000 trials). Relearning nodes that received new patterns then takes several hours even on a Sun SPARCstation 10/31. This gave rise to a suggestion of Groen [**?**], to use a single-layered perceptron instead of a feed-forward network to implement an approximator.

4.1.1 General least squares

A perceptron attempts to fit a hyperplane to a set of known points in that hyperplane using linear interpolation. The method used by us uses a generalisation of linear regression, a method that fits one-dimensional data to a straight line.

Let us consider a set of N data points (x_i, y_i). We want to fit these points to a straight-line model

$$y(x) = ax + b$$

by varying a and b. The uncertainty associated with each measurement y_i is called σ_i and is assumed to be known. To measure how well the model agrees with the data, the following chi-square merit function is used:

$$\chi^2(a,b) = \sum_{i=1}^{N} \left(\frac{y_i - b - ax_i}{\sigma_i} \right)^2 \tag{4.1}$$

This equation is minimized to determine a and b. In our case, all data points generated are correct since only correct $\Delta\theta$ values

$$(\Delta\theta_1, \Delta\theta_2, \Delta\theta_3) = \mathcal{F}(\vec{i})$$

are generated, i.e., the data points are part of $\mathcal{F}$. Therefore, all σ_i values are equal and for convenience set to 1. Another distinction is the dimensionality. A best fit of a hyperplane is wanted that approximates a multi-dimensional set of data points rather than a set of one-dimensional data points. The perceptrons have to deal with a set of N data points $(x_{1i}, \ldots, x_{ni}, y_i)$ that we want to fit in an n dimensional plane

$$y(x_1 \ldots x_n) = a_1 x_1 + \cdots + a_n x_n + b \tag{4.2}$$

The y value of equation (??) represents a desired $\Delta\theta$ value for one of the joints of the robot. The input vector, describing the input retrieved from the camera $\vec{c} = (c_x, c_y, c_z)$ and the robot position (θ_2, θ_3) represent $x_1, \ldots, x_5$. The $\Delta\theta$ values of three joints are wanted, so three equations similar to equation (??) exist, one for each joint. The equations differ in the values of $(a_1, \ldots, a_5, b)$, which describe the individual hyperplane of a joint.

Equation (??) is rewritten to represent n dimensional data; the chi-square merit function equals

$$\chi^2(a_1 \ldots a_n, b) = \sum_{i=1}^{N} (y_i - b - a_1 x_{1i} - \cdots - a_n x_{ni})^2 \tag{4.3}$$

This equation is minimized to determine

$$a_1, \ldots, a_n, b.$$

At its minimum, derivatives of $\chi^2(a_1, \ldots, a_n, b)$ vanish:

$$\begin{aligned} 0 = \frac{\partial \chi^2}{\partial b} &= -2 \sum_{i=1}^{N} (y_i - b - a_1 x_{1i} - \cdots - a_n x_{ni}) \\ 0 = \frac{\partial \chi^2}{\partial a_j} &= -2 \sum_{i=1}^{N} (x_{ji}(y_i - b - a_1 x_{1i} - \cdots - a_n x_{ni})) \end{aligned} \tag{4.4}$$

with $1 \leq j \leq n$. These conditions can be written down in a convenient form if the following sums are defined.

$$\begin{aligned} S &\equiv \sum_{i=1}^{N} 1 \\ S_{x_j} &\equiv \sum_{i=1}^{N} x_{ji} && 1 \leq j \leq n \\ S_{x_j x_k} &\equiv \sum_{i=1}^{N} x_{ji} x_{ki} && 1 \leq j,\ k \leq n \\ S_y &\equiv \sum_{i=1}^{N} y_i \\ S_{x_j y} &\equiv \sum_{i=1}^{N} x_{ji} y_i && 1 \leq j \leq n \end{aligned}$$

With these definitions equation (??) becomes

$$\begin{array}{lll} S_y & = bS + a_1 S_{x_1} & + \cdots + a_n S_{x_n} \\ S_{x_1 y} & = bS_{x_1} + a_1 S_{x_1 x_1} & + \cdots + a_n S_{x_1 x_n} \\ S_{x_2 y} & = bS_{x_2} + a_1 S_{x_2 x_1} & + \cdots + a_n S_{x_2 x_n} \\ & \vdots & \\ S_{x_n y} & = bS_{x_n} + a_1 S_{x_n x_1} & + \cdots + a_n S_{x_n x_n} \end{array}$$

These are $n+1$ equations with $n+1$ unknowns $a_1, \ldots, a_n, b$. If n will be assumed to be 5, i.e., the dimensionality of $\mathcal{F}$, then the above equations are equivalent to the matrix equation

$$\begin{pmatrix} S_{x_1} & S_{x_2} & S_{x_3} & S_{x_4} & S_{x_5} & S \\ S_{x_1x_1} & S_{x_1x_2} & S_{x_1x_3} & S_{x_1x_4} & S_{x_1x_5} & S_{x_1} \\ S_{x_1x_2} & S_{x_2x_2} & S_{x_2x_3} & S_{x_2x_4} & S_{x_2x_5} & S_{x_2} \\ S_{x_1x_3} & S_{x_2x_3} & S_{x_3x_3} & S_{x_3x_4} & S_{x_3x_5} & S_{x_3} \\ S_{x_1x_4} & S_{x_2x_4} & S_{x_3x_4} & S_{x_4x_4} & S_{x_4x_5} & S_{x_4} \\ S_{x_1x_5} & S_{x_2x_5} & S_{x_3x_5} & S_{x_4x_5} & S_{x_5x_5} & S_{x_5} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ b \end{pmatrix} = \begin{pmatrix} S_y \\ S_{x_1y} \\ S_{x_2y} \\ S_{x_3y} \\ S_{x_4y} \\ S_{x_5y} \end{pmatrix} \tag{4.5}$$

This equation is solved using the *LU-Decomposition* technique. This technique manipulates the matrix such that solving the equation is possible using back substitution, which can be performed efficiently. Decomposing an $N \times N$ matrix requires $\frac{1}{3}N^3$ executions of a multiply and an add instruction. Once the matrix has been decomposed, solving becomes a complexity $O(N^2)$ operation using back-substitution. For a more detailed discussion of the LU-Decomposition technique refer to [?, Section 2.3]. The main advantage of the LU-Decomposition method is that back-substitution can be used with arbitrary right-hand sides in the equation. This proves to be very useful considering that there are three joints and therefore three equations similar to (??). The equations of each joint only differ in the produced y value, which is part of the right-hand side of (??) only. The matrix itself remains the same for each joint, since no y values are to be found in the matrix. The three equations can be solved by decomposing the matrix once and performing back-substitution three times, once for each joint.

4.1.2 Incremental property

From equation (??) it can be seen that the matrix and the right-hand side consist solely of sums. A property of the sums used here is that they can be updated *incrementally*. The subscript of a sum S indicates what value has to be added to S to update it. For instance, to update sum $S_{x_px_q}$ for learning pattern i one would perform

$$S_{x_px_q}[i] = S_{x_px_q}[i] + x_{pi}x_{qi}$$

This is called an incremental update of $S_{x_px_q}$. Consequently, updating the entire matrix and the right-hand side can be done at relatively little computational cost.

4.2 Attaching bins to the tree

As was mentioned in section ?? the Nested Perceptron Approach and the Nested Network method each use a different method to attach bins to the tree. The former used virtual nodes to create a path to a leaf in the tree, and attached a bin at that leaf. The advantage of this method is that patterns are always stored in the correct place, since their leaf represents exactly the same subspace as which the bin represents. Consequently, the bins do not have to be moved during

the learning process. Retrieving patterns for a particular subtree can then simply be done by recursively searching that subtree for bins; any bin found will contain patterns that belong to the subspace that the subtree represents.

The drawback of this method is, as was noted earlier, the huge amount of memory resources it requires. Virtual nodes may take up as much as 25 megabytes of memory during one run of only 1,500 trials. Since it was expected that the usage of perceptrons would require a deeper tree to establish the same precision as the neural network approach, the memory requirements for virtual nodes would have become unreasonably large. Also, using a perceptron as an approximator as opposed to a feed-forward network as was previously done, it was expected that more trials, and hence more learning patterns and bins would have to be created to reach the same precision as was attained with feed-forward networks. This would lead to a larger tree and even more memory consumption.

It is clear from this that a way had to be found to store patterns such that on the one hand virtual nodes would be no longer required, and on the other hand to avoid the need of large amounts of computational resources to search and store learning patterns. A typical run of 1,500 trials creates as much as 40,000 bins, holding about 50,000 patterns. This called for the search of an approach that can handle these amounts of bins and patterns in a manner that keep computational resources as well as memory requirements at an acceptable level.

The problem has been solved in the following way. Each learning pattern $\vec{l} = (\vec{i}, \vec{t})$ has to be stored in a bin having a unique identification

$$\langle b_1, b_2, \ldots, b_D \rangle \tag{4.6}$$

where b is a branch number and D is the maximum depth of the tree. (See also section **??**). The general idea of the solution is that a bin with identification $\langle b_1, b_2, \ldots, b_D \rangle$ should always be linked to the deepest existing node $N[d]$, representing a subspace

$$U_{\langle b_1, b_2, \ldots, b_D \rangle} \subseteq U_{N[d]} \tag{4.7}$$

Note that for any given tree, exactly one node applies. To accomplish this, the following actions are taken when a learning pattern has to be stored:

- Calculate the identification of the bin that should receive the learning pattern $\vec{l}$ (using $\vec{i}$),
- Descend the tree, starting at the root node and following the branch numbers $\langle b_1, b_2, \ldots, b_D \rangle$ until a leaf node $N[i]$ is encountered,
- Sequentially search the linked list of bins connected to $N[i]$ for a bin that has the same identification as the learning pattern. If such a bin is not found, create it and link it to the end of the list of bins that is connected to $N[i]$,
- Store the pattern in the found or newly created bin.

A split can only occur if, among other things, enough patterns exist for the to be created subspace (see section **??**). Since bins have to be linked to the deepest possible node that represents a space which the bin represents a subspace of (see equation (**??**)), there will be bins that have to be moved after a split occurred. These bins are searched and linked to the newly created node by rearranging pointers of the single linked list of bins. This process will hence be referred to as *bin-sinking*. See figure **??** for a visualisation of this process. The consequence of this method is that the list of bins for any given node will always be relatively short. For instance, at the point that a node has been entirely split (i.e., all its subspaces are represented by a child node), the list of bins connected to that node is empty, since all bins have been moved to the child nodes. It will

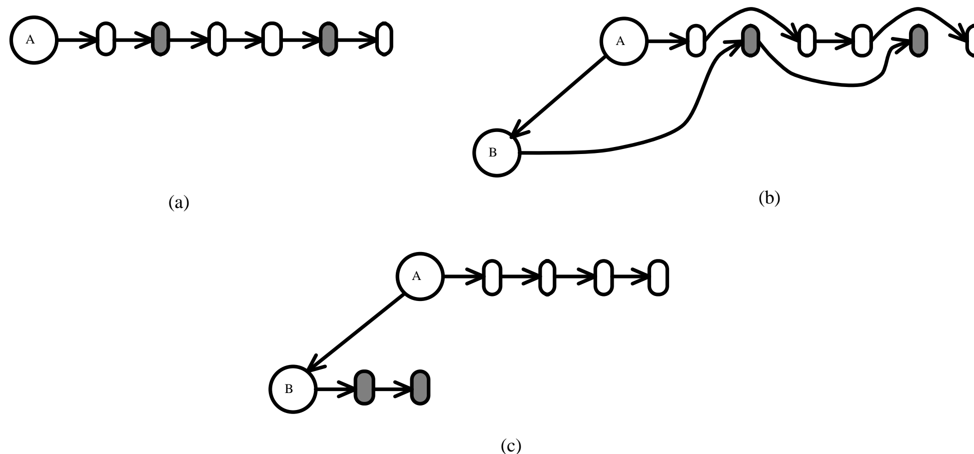


Figure 4.1: Bin-sinking; moving bins after a split. Consider node A that is going to be split (a). A linked list of bins is attached to this node, they represent subspaces of node A. The gray bins in (a) lie inside the subspace that will receive a new node B. Those bins have to be linked to the new node B. This is done by rearranging the pointers to the bins appropriately (b). The result is depicted in (c).

never receive a bin in this situation as a consequence of the way learning patterns are stored (see above).

Collecting learning patterns in order to determine if a node $N[d]$ needs to be split at one of its subspaces $U_{N[d+1]} \subset U_{N[d]}$, can be done by searching the list of bins linked to $N[d]$ and selecting learning patterns $\vec{l} \in U_{N[d+1]}$. This can be done as follows. As noted above, every bin has a unique identification $\langle b_1, b_2, \ldots, b_D \rangle$. For bins linked to node $N[d]$, a node at depth d, applies that $b_1, \ldots, b_d$ are identical. To find bins that represent subspace $U_{N[d+1]}$ only b_{d+1} has to contain the correct branch number of the subspace. The list of bins connected to $N[d]$ can therefore be searched very quickly for bins representing $U_{[d+1]}$. In addition to the fact that the list of bins will never be very long because of bin-sinking, the computational cost for retrieving patterns for any given subspace is very low, as well as the computational cost for searching a particular bin. This is important because these actions occur very often, the latter every time a learning pattern is stored.

When a merge of a node occurs, all bins are moved back to the parent node. More precisely, the list is linked to the end of the list of bins of the parent node. See figure **??** for a visualisation of this process.

It should be clear from both figures **??** and **??** that bins are not actually moved, but that only pointers are rearranged. This also helps to keep the computational cost very low.

4.2.1 Using multiple perceptrons for one approximator

The incremental property of perceptrons (see section **??**) has one major disadvantage. The implementation of perceptrons as described there will not discard any learning pattern it has learned. Gradually, individual learning patterns will have less and less influence on the output of an approximator. At the time a very large number of learning patterns have been learned to any given approximator, additional learning patterns will hardly have any effect on the output of that approximator. This is caused by the sums in equation (**??**) which become very large. Adding a new learning pattern to large sums will have only little influence on the solution; $a_1, a_2, \ldots, a_n, b$

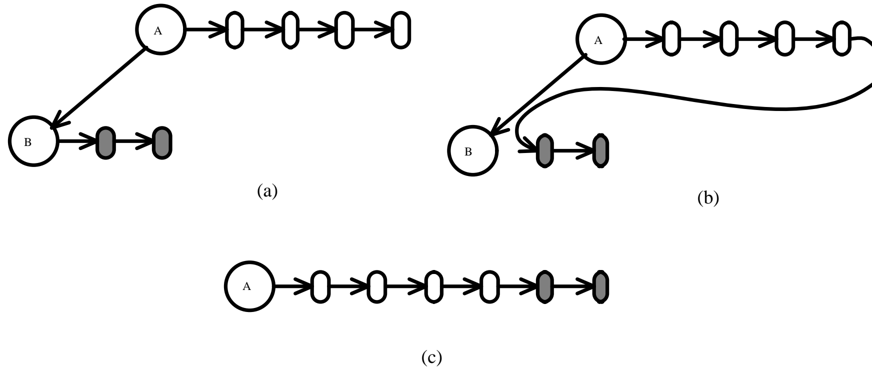


Figure 4.2: Moving bins after a merge. Consider node B that is going to be merged (a). Its bins, colored gray for convenience, are moved back to the parent node of B by just linking them at the end of the list of bins connected to node A (b). Lastly, node B is deleted (c).

will hardly change when the sums in equation (**??**) are very high. More precisely, the change in the solution approaches zero when S approaches infinity.

The consequences of these observations are disastrous when minor chances in $\mathcal{F}$ occur. If changes in $\mathcal{F}$ are sufficiently small so that the approximator is not merged, but still produce a considerable error, then it would take a very large number of learning patterns for the approximator before it enhances its performance and adapt to the change in $\mathcal{F}$. However, it would never adapt entirely since it does not discard those now invalid learning patterns it learned prior to the change in $\mathcal{F}$. Therefore a facility that would gradually discard older learning patterns is necessary. There are several different ways to accomplish this. One could for instance introduce an α that would define how quickly old learning patterns are forgotten. This α would adapt the sums in equation (**??**) such that the influence of older patterns on the solution will gradually diminish. Another approach is to store the n most recently arrived learning patterns for any node, and subtract those learning patterns one by one from the moment $n+1$ patterns have been taught to a perceptron, such that a perceptron will represent the n most recently arrived learning patterns. The downside of this approach is that many learning patterns will have to be stored which places great demands on memory resources.

The solution chosen was to simply create a multiple number of perceptrons for each approximator. The user can specify how many perceptrons an approximator should use and the maximum number of learning patterns a perceptron may contain. These two parameters define how quickly learning patterns should be discarded, as will be explained below. The benefits of this approach are first of all a modest increase of memory and computational cost. A perceptron takes only a few additions and multiplications to add a learning pattern, and the solution only has to be computed for one perceptron. Secondly, using multiple perceptrons creates the possibility to detect changes in $\mathcal{F}$ by calculating and examining the solution of each of the perceptrons. If this solution suddenly shows large changes, this may be reason to believe that a change in $\mathcal{F}$ has occurred. In the current implementation however this beneficial characteristic of multiple perceptrons has not yet been exploited.

It may be helpful to take a look at figure **??** before reading the following explanation of perceptrons. Let p be the number of perceptrons per approximator and let q be the maximum number of learning patterns p may contain. The perceptrons are linked to the node by a circular linked list. When the learning process starts, only the first perceptron of an approximator receives learning patterns. As soon as q/p patterns have been learned to the first perceptron, the next perceptron also starts receiving learning patterns. At that point, two perceptrons are updated

simultaneously. However, the solution represented by $(a_1, a_2, a_3, a_4, a_5, b)$ of equation (**??**) is still being calculated using the first perceptron. After every q/p patterns, a new perceptron starts to receive learning patterns for the first time. After q patterns have been trained to an approximator, all p perceptrons are receiving learning patterns. At that point, the first perceptron is being cleared, and the second perceptron will be used to calculate the solution for the approximator. Because of this switch to the new perceptron, q/p learning patterns are discarded, since the second perceptron started to receive learning patterns after q/p patterns had been taught to the first perceptron. The first perceptron that just has been cleared, and which contained q learning patterns, will start receiving new learning patterns.

With this mechanism, the user can manipulate the pace in which learning patterns should be discarded, and how many at a time. An ideal situation exists when $p = q$ since then for every new learning pattern that arrives, an old learning pattern is discarded. However, this is not feasible regarding the memory and computational cost this would take for large a q. Furthermore, it has been experimentally established that a p of below 20 always suffices, even for a large q.

The consequences are that when a minor change in $\mathcal{F}$ occurs, the approximator will have adapted to the new situation after at most q learning patterns have been trained to that approximator.

4.3 Avoiding the Input Adjustment Method

Using the input adjustment method is a convenient way to estimate correct points in $\mathcal{F}$ as long as the error is linear. It is possible to avoid the input adjustment method and replace it by a method which can also handle systems that have a non-linear error. Also, this replacement is more general and elegant than the input adjustment method, which inherits problem specific elements. However, the this replacement also has some drawbacks which will be discussed also.

4.3.1 Modifications concerning learning patterns

Recall that a learning pattern $\vec{l}$ consists of a camera output vector $\vec{c}$, a current robot state, and a target vector $\vec{t}$:

$$\vec{l} = (\vec{i}, \vec{t}) = (c_x, c_y, c_z, \theta_2, \theta_3; \Delta\theta_1, \Delta\theta_2, \Delta\theta_3)$$

The desired delta joint values $\vec{t}$ are equal to the delta joint values actually generated. In order to produce a correct learning pattern, the input is adjusted such that $\vec{t}$ represents the desired movement from the adjusted input vector and the target position.

In order to avoid adjusting the input, we add another vector, $\vec{d}$ which represents the x, y and z values retrieved from the camera image frame *after* movement $\vec{t}$, as opposed to $\vec{c}$, which represents the x, y and z values *before* movement. With this additional vector, an alternative learning pattern can be constructed:

$$\vec{l} = (\vec{i}, \vec{t}) = (c_x, c_y, c_z, \theta_2, \theta_3, d_x, d_y, d_z; \Delta\theta_1, \Delta\theta_2, \Delta\theta_3)$$

This is a valid learning pattern, since it describes a camera output vector $\vec{c}$ before movement, a robot position θ_2, θ_3 before movement, a camera output vector $\vec{d}$ after movement, and the movement itself, $\Delta\vec{\theta}$. Informally, it describes what movement should be made to change the camera output vector from $\vec{c}$ to $\vec{d}$ for a given robot position. The goal of the system is to have $\vec{d} = (0, 0, 0)$, i.e., to have the end-effector at the target position:

$$\mathcal{N}(c_x, c_y, c_z, \theta_2, \theta_3, 0, 0, 0) \rightarrow \Delta\vec{\theta}$$

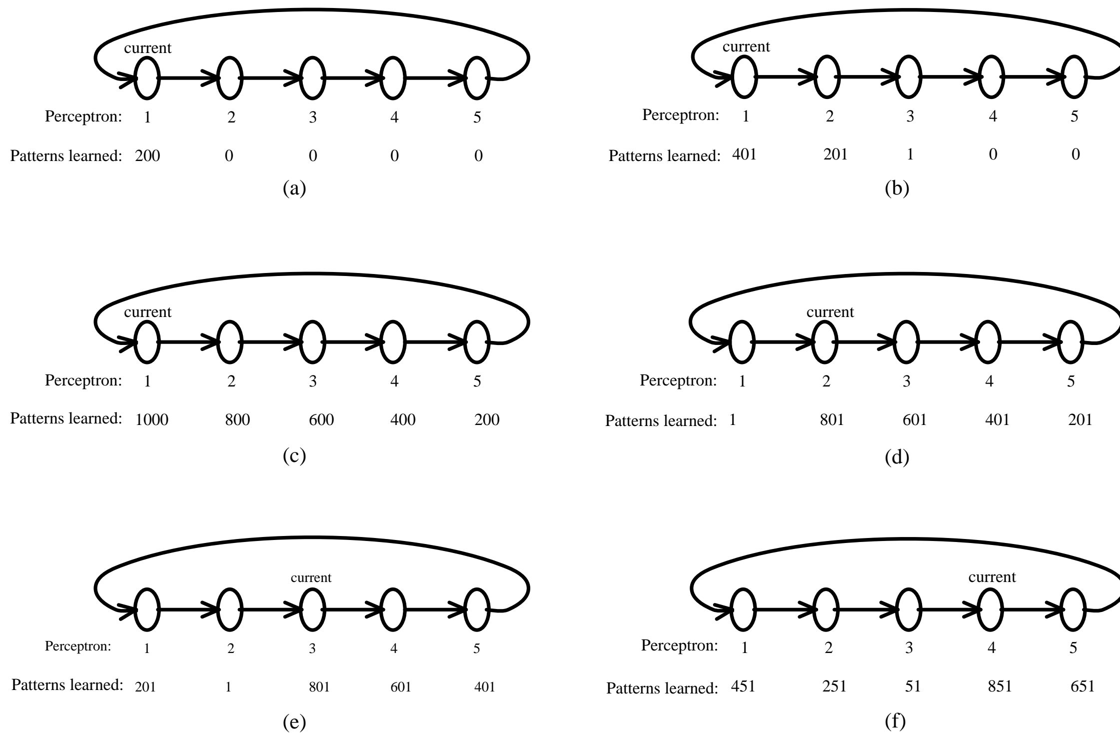


Figure 4.3: Example of multiple perceptrons linked to a node. This example uses $p = 5$ and $q = 1000$, i.e., 5 perceptrons which can contain at most 1,000 learning patterns. The word 'current' in the picture denotes which perceptron is used to calculate the solution for $\mathcal{N}$. The initial $1,000/5 = 200$ learning patterns will be taught to the first perceptron only (a). Learning patterns 201 up to 400 are taught to perceptrons 1 and 2, learning patterns 401 to 600 to perceptrons 1, 2 and 3, (b) and so on. Learning patterns 801 to 1,000 finally will be taught to all five perceptrons, but the solution is still being calculated using the first perceptron, since it has not exceeded the limit of 1,000 learning patterns yet (c). However, at learning pattern 1,001 perceptron 1 is cleared, and the solution will be calculated using perceptron 2, which has learned 801 patterns at that point (d). When 1000 learning patterns have been taught to perceptron 2, it is cleared and the solution will be calculated using perceptron 3 (e). Finally, (f) pictures some moment in time in which perceptron 4 is in use.

The system is asked to produce a movement such that $\vec{d} = (0,0,0)$. After the movement, $\vec{d}$ is measured again, and the learning pattern will be altered: the three zeroes will be replaced with the actual $\vec{d}$ in order to produce a correct learning pattern, which can then be trained to an appropriate approximator.

4.3.2 Partially implicit subspace assignment

This method increases the dimensionality of $\mathcal{F}$ from five to eight since there now are eight inputs. Nevertheless, the maximum number of children of a node can remain $2^5 = 32$. Since three dimensions have been added to the input space, one might expect that the maximum number of children of a node would become $2^8 = 256$. This is not necessary, since the input patterns that are fed to $\mathcal{N}$ have a fixed $\vec{d} = (0,0,0)$. For this reason approximators only need to represent subspaces that intersect with $\vec{d} = (0,0,0)$ which is only true for 32 of the 256 possible partitions. The learning patterns that are created however have an arbitrary $\vec{d}$ that need to be represented in order to be able to interpolate for $\vec{d} = (0,0,0)$. Therefore the root node not only represents the entire input space, but also every $\vec{d}$ that can possibly be generated. Since nodes at deeper levels generally show improved representation of a subspace, the generated error will be lower. This translates into learning patterns that will have $\vec{d}$ closer to $(0,0,0)$, since $\|\vec{d}\|$ is equivalent to the distance between the position the end-effector actually reached and the desired position. This property can be used in a beneficial way by narrowing the input space of $\vec{d}$ implicitly for nodes at deeper levels by adjusting the boundaries such that they will be closer to $(0,0,0)$. Mathematically, consider the situation in which the root node represents the entire input space and $\vec{d}(x,y,z)$ is bounded by

$$-a_x < d_x < a_x; \quad -a_y < d_y < a_y; \quad -a_z < d_z < a_z$$

For an approximator at depth d applies that it will represent vectors $\vec{d}$ that are bounded by

$$-a_x n^d < d_x < a_x n^d; \quad -a_y n^d < d_y < a_y n^d; \quad -a_z n^d < d_z < a_z n^d; \quad 0 < n \leq 1$$

Informally, the boundaries around zero for some node $N[d]$ are narrowed by a constant amount n for node $N[d+1]$. It is referred to as implicitly narrowing the input space as opposed to explicitly, because it is not clear from the branch numbers of a node what the boundaries around $(0,0,0)$ for a particular node at depth d are. They are implicitly defined by the depth of that node. In addition to that, it is implicitly true that every of the 32 partitions intersect with $\vec{d} = (0,0,0)$. The narrowing constant, n, is user-definable.

4.3.3 Modifications concerning branch numbers

Using the discussed scheme to represent the eight dimensional $\mathcal{F}$ called the need for a modification in the way branch numbers of learning patterns and bins are assigned. Recall that the identification of bins and learning patterns consists of branch numbers which describe a path to a leaf in the tree (see section **??**). Since the root node represents the entire input space, every possible learning pattern and bin can be represented by the root node. The method used to build the tree as described in section **??** has the property that all nodes at some depth d will together cover the entire input space. The introduction of the partially implicit assignment of input space combined with the narrowing constant n has as a consequence that the aforementioned property does no longer apply for any $n < 1$. Consider a learning pattern $\vec{l} \in U_{N[d]}$ and a node $N[d+1]$ such that $U_{N[d+1]} \subset U_{N[d]}$. Assume that for $n = 1$ applies $\vec{l} \in U_N[d+1]$. The situation now can occur that for some $n < 1$ holds that $\vec{l} \notin U'_{N[d+1]} \subset U_{N[d+1]}$. Such learning patterns can never reach a node at depth $d+1$. If $n = 1$ every learning pattern can end up at the maximum depth D in the tree.

If $n = 0$, just to take the other extreme, only the root node can represent learning patterns. There would never occur a split, since nodes at a depth > 1 represent subspaces of the input space of size zero.

It should be clear from this that a special case concerning branch numbers had to be introduced to represent the possibility that none of the partitions of a node $N[d]$ applies for a learning pattern $\vec{l} \in U_{N[d]}$, and consequently, there does not exist a branch number such that $\vec{l} \in U_{N[d+1]} \subset U_{N[d]}$. This has been solved by assigning a special type of branch number which actually does not represent a valid branch number, but instead indicates that no branch number applies. This type of branch will be denoted by $\bar{b}$. Recall that an identification of a bin or learning pattern has been defined as

$$\langle b_1, b_2, \ldots, b_D \rangle$$

For the learning pattern $\vec{l}$ mentioned earlier, applies:

$$\langle b_1, b_2, \ldots, b_d, \bar{b}_{d+1}, \ldots, \bar{b}_D \rangle$$

Note that as soon as there does not exist a subspace for some learning pattern $\vec{l} \in U_{N[d]}$ at depth $d + 1$, branch numbers $d + 1 \ldots D$ will contain $\bar{b}$. While there are 32 partitions numbered from 0 to 31, by definition the number 32 has been assigned to represent $\bar{b}$.

Since every learning pattern is stored, so are learning patterns that have reached the maximum depth they can possibly reach in the tree. These are learning patterns with identification

$$\langle b_1, b_2, \ldots, b_d, \bar{b}_{d+1}, \ldots, \bar{b}_D \rangle \tag{4.8}$$

that have reached depth d in the tree. For instance, a learning pattern that can never reach depth 1 in the tree looks like

$$\langle \bar{b}_1, \bar{b}_2 \ldots, \bar{b}_D \rangle$$

These learning patterns are stored in the appropriate bin, even though they could just as well been thrown away, since they will never be used again. However, for simplicity, they are not. Every bin can hold a maximum number of learning patterns. If a bin is full and a new learning pattern has to be stored in that bin, the oldest learning pattern will be discarded in favor of the new learning pattern. Because of this mechanism, the number of learning patterns as in equation (**??**) that are stored for any node at depth d will be at most the number of learning patterns one bin can maximally hold, which is typically less than 10, a very low number. In other words, the number of unused learning patterns stored by the system will be at most the number of nodes created multiplied by the maximum number of learning patterns a bin can hold. This slight waste of memory is bearable.

4.3.4 Consequence on performance

Not using the input adjustment method and increasing $\mathcal{F}$ to eight dimensions has consequences on the precision and performance of the system. In the first place, a perceptron now consist of 9×9 matrix, which uses more computational resources to solve as well as larger memory requirements to store (see also section **??**). Secondly, approximating an eight dimensional input space is much more difficult as approximating a five dimensional one. In chapter **??** these differences are tested.

4.4 Problems with the split and merge algorithm

While testing the relative and absolute merging algorithms it appeared that those methods have very limited usability, despite their seemingly straightforward, sound and simple definition. To

understand why this is the case, one should keep in mind that the one and only goal of merging parts of the tree is to fully recover from a change in $\mathcal{F}$ and regain the same precision of the approximation of $\mathcal{F}$ as was accomplished prior to the change in $\mathcal{F}$. This recover should take place in as few trials as possible. In the following sections a closer look will be taken at how the tree is split up during the learning process, and how it is broken down when a change in $\mathcal{F}$ occurs. Throughout this section it should be kept in mind that many user-definable parameters influence the behavior of the merge algorithm. In some cases directly, like the split error, the relative merge error and the absolute merge error, as well as the number of learning patterns that is used for averaging the error. In other cases indirectly, like the number of matrices that a perceptron consists of, and the number of learning patterns that a matrix can contain (as explained in section **??**).

4.4.1 Behavior of the split algorithm

The split algorithm was designed such that parts of the input space that are hard to approximate are split more often than parts of the input space that are easy to approximate. However, parts of the input space that receive a relatively large number of learning patterns will also be split more often. This happens because of the fact that the more learning patterns a particular subspace receives, the more likely it is that parts of that subspace are found that produce a high enough error for a split. Stated otherwise, one could say that if a subspace receives relatively many learning patterns, that subspace is examined more thoroughly than other spaces. Hence it is likely to find subspaces with a relatively high split error. For the particular case of the eye-hand mapping function, one will find parts of the input space that represent an area that lie close to the center of the image frame, or more precisely, close to $\vec{c} = (0, 0, 0)$ to be split relatively often. This part of the input space receives most learning patterns, because the robot will visit this part of the input space almost each trial as soon as the system has been trained sufficiently, since it then is able to approach the target position closely at every trial.

4.4.2 Merging nodes wrongly

At all cost it should be avoided that parts of the subtree are merged wrongly, i.e., merged before a change in $\mathcal{F}$. The performance of the system almost always drops considerably when nodes are merged wrongly anywhere in the learning process. The system builds a tree consisting of about 250 up to 350 nodes within 1,500 trials. A merge throws away complete subtrees, because of the nature of the merge algorithm. It is assumed that as soon as a node has been found that has a parent performing better, not only the child node itself but also its children perform worse than its parent. It therefore will seriously damage the accuracy if for instance at any point in the learning process 30 nodes are just thrown away wrongly. It can take hundreds of trials before these nodes are restored again. If nodes are being merged wrongly every once in a while during the learning process before a change in $\mathcal{F}$ has occurred, the performance of the system will never even come close to its maximal potential.

4.4.3 Limitations of the absolute merge algorithm

Let us look more closely at the absolute merge mechanism. Every approximator has an efficiently implemented bookkeeping mechanism that incrementally updates the average error over some user-definable number of learning patterns. It is necessary to take an average, typically over the 300 to 500 most recently arrived learning patterns, in order to avoid outliers in the error. Would this not be done, or would the number of learning patterns that is used for the average be too low, the error becomes susceptible to peaks of individual learning patterns which could cause a node to be merged wrongly.

Each time a learning pattern arrives in a node, the absolute error of that node is updated to reflect the current situation. However, in fact the actual error in all approximators instantly jump to a higher value at the very moment $\mathcal{F}$ changes, but it takes a considerable number learning patterns to detect this due to the averaging. Since all learning patterns are propagated from a leaf in the tree upwards to the root, the root will receive all learning patterns, while deeper levels in the tree will receive much less patterns. Consequently, when $\mathcal{F}$ changes dramatically, the error in the root node will significantly raise, and this will happen more quickly than in any other node. The root node will therefore be likely to be the first node to produce an error that is higher than the treshold value that is used to decide if an absolute merge should be executed. As soon as the root node is being merged, the entire tree will be thrown away and the system will learn from scratch. This demonstrates the limited usability of the absolute merge algorithm, since it is only useful to detect a major change in $\mathcal{F}$. It can not be used to detect minor changes, since the height of the treshold value should be such that it is only exceeded when a major change occurs. It is not possible to lower the treshold value such that it would also detect minor changes because the absolute error fluctuates quite a bit. It is very undesirable when the entire tree would be merged wrongly in the middle of the learning process, just because an unfortune fluctuation in the absolute error in the root node makes it exceed the treshold value. In addition to this, the system should be designed such that it can recover from minor changes without having to relearn $\mathcal{F}$ from scratch. The bottom line is that the absolute merge mechanism is only useful to detect a major change in $\mathcal{F}$.

4.4.4 Limitations of the relative merge algorithm

Now let us look at the relative merge mechanism. The idea behind it seems simple and straightforward as well: just merge a node if one of its ancestors performs better. This is based on the fact that nodes at deeper levels should have a better representation of their subspace than nodes at higher levels, since nodes at deeper levels represent a smaller area of the subspace. Differently put, nodes at higher levels receive more learning patterns than nodes at deeper levels and are therefore able to adapt more quickly to a minor change in $\mathcal{F}$ than nodes at deeper levels. One would expect to see a relatively high error in nodes at deeper levels and a relatively low error in nodes at higher levels shortly after a minor change in $\mathcal{F}$. There are two ways to implement the merge algorithm that uses these properties, which are the wrong way and the very wrong way. The very wrong way is to compare absolute errors of a node and its ancestors and merge that node when its absolute error is higher by some treshold value as its ancestor node. This is obviously wrong since it is not be true *in general* that nodes at deeper levels show a lower error than nodes at higher levels. For instance, a child node of some node $N[d+1]$ may be assigned to a relatively difficult part of the input space. Although its parent node $N[d]$ also has to approximate $U_{N[d+1]} \subset U_{N[d]}$, it approximates a much larger part of the input space that may be not so hard to approximate. Therefore it is not unlikely that in some cases a node at a deeper level shows an higher average absolute error than one of its ancestors, caused because it has to represent a difficult part of the input space. The aforementioned scheme would cause nodes that happen to represent difficult parts of the input space to be merged, which obviously is a very undesirable effect.

An enhancement of this scheme is the relative merge mechanism as described in section **??**. This mechanism avoids the problems just mentioned by selecting learning patterns that belong to $U_{N[d]}$ and calculate the error those patterns produce in their ancestors. Using this scheme, it can no longer occur that a node $N[d+1]$ is merged wrongly just because it is compared with its parent node $N[d]$ that happens to produce a lower error for its input space $U_{N[d]} \supset U_{N[d+1]}$, while it might still produce an higher for $U_{N[d+1]}$. Now with this problem out of the way, the relative merge still does not work satisfactory. In order to understand this, it is necessary to comprehend many of the details of the behavior of the error.

Consider the situation in which a change in $\mathcal{F}$ occurs, for instance because one of the links is elongated by 10 centimeters. Assume that this change in $\mathcal{F}$ occurs at a moment when the tree is built up out of more than a hundred nodes, i.e., when the learning process is well on its way. Ideally, one should see the following behavior. Since nodes at higher levels receive many learning patterns, they will recover first from this change in $\mathcal{F}$. As soon as nodes at higher levels are recovered, the lower levels in the tree are merged because their error will remain higher than the error at the higher levels. This happens because the lower levels receive fewer learning patterns and therefore will not be able to adapt to a minor change in $\mathcal{F}$ as quickly as nodes at higher levels. The behavior should have the effect that nodes which adapt quickly are not merged, while nodes that do not adapt quickly are merged. The idea behind this is that the system should fall back to a more coarse approximation of $\mathcal{F}$ when a small change occurs and will gradually rebuilt its fine approximation by rebuilding the deeper levels of the tree. This scheme is better than throwing everything away and start learning from scratch, since the latter will require more trials, because it also has to rebuild the coarse approximation. After a small change in $\mathcal{F}$, the coarse approximation adapts itself to the new situation in only a few trials because of the many learning patterns higher levels in the tree receive.

Unfortunately, this ideal behavior described above will not be achieved. As described previously, the relative merge error is a treshold value. If the performance of a child node $N[d+1]$ is significantly worse than its parent node $N[d]$ for the same part of the subspace, the child node is merged (see also equation (**??**)). The relative merge error should be such that nothing is merged as long as no change in $\mathcal{F}$ occurs. This may sound trivial, but it is not. Because of the linear interpolation algorithm that is used, any perceptron will have an optimal representation of its learning patterns that have been taught to it. There can not exist a better least-mean-square solution for those learning patterns. Hence, it impossible that a parent node has a better representation of all learning patterns in a child node than the child node itself. There will never be a moment in the learning process where a parent node produces a lower error for *all* learning patterns of a child node than the child node itself, since the child node has calculated the optimal solution. This remains true even after any change in $\mathcal{F}$. Would all learning patterns of a child be used to decide which node produces the lowest error for those patterns, the child node would always win, and consequently no relative merged would ever be executed. Therefore, in order for the relative merge be able to work, the error of a subset of learning patterns is selected. This subset consists of the n most recently arrived learning patterns that have arrived in the child node. The average error is calculated for the child node, which is the error of that child node, as well as for the parent node, which is the error of the parent node when it approximates the subspace of its child node. Using this scheme, the property that nodes at higher levels adapt more quickly to a change in $\mathcal{F}$ than nodes at deeper levels can be exploited. Consider what happens when a minor change in $\mathcal{F}$ occurs. Although the child node will still represent *all* of its learning patterns better than the parent node by definition, it might not do so for the n most recently arrived learning patterns. After the minor change in $\mathcal{F}$, the parent node will adapt itself more quickly to the new $\mathcal{F}$ since it receives more learning patterns. Even though these patterns are from different subspaces as that of a particular child node, they help adapting the parent node to the change in $\mathcal{F}$. Any child node will receive less patterns and will not be able to adapt as quickly as its parent node. At a certain point, when n most recently arrived learning patterns that belong to the subspace of some child are both propagated through the child as well as through the parent, the parent might show lower error than the child and the child node can be merged.

The chance that a node is merged wrongly using this scheme might be very small, but it is far from neglectable. It is not impossible for a parent node to represent the most recently n learning patterns of its child node better than the child node itself. The chance may be small, but with the many nodes that a tree consists of, and the many learning patterns that are created, it occurs.

That is why a treshold value was introduced, called the merge error (ϵ_r in equation (**??**)). This treshold value should prevent that nodes are merged when no change in $\mathcal{F}$ has occurred, which should be avoided at all cost, as was explained in section **??**. However, at the same time this treshold value can prevent a subtree from being merged even when a merge would be appropriate. This behavior will show some very undesirable effects. Again, consider a parent-child node pair. Suppose $\mathcal{F}$ has been altered some trials earlier, and the parent node starts to produce a lower error for the learning patterns of the subspace of its particular child, but not low enough for the child to be merged. The error in the child node will rise above the split error for many parts of its subspaces, and the child node will start splitting up where possible. The effect is that the number of nodes after a small change in $\mathcal{F}$ suddenly starts to raise considerably. Consider the possibility that this child node resides at a fairly low level in the tree and receives only few learning patterns. In that case, it will not be split for other parts of its subspace for there might not exist enough learning patterns within that subspace in order to split it (section **??**). When the system requests a joint displacement for that particular part of the input space, a huge error can be expected to be generated, the robot will probably move well away from the target if it was already close. After a minor change in $\mathcal{F}$, a considerable number of pieces of input space that show this unwanted behavior will emerge, making the approximation of $\mathcal{F}$ rather unreliable. One could propose to just learn on until this kind of nodes have regained their normal precision, relying on the adaptive nature of multiple perceptrons. This however will take an unreasonable number of trials and learning patterns, since those nodes often reside at low levels in the tree and receive only few learning patterns. Also, because of the change in $\mathcal{F}$, notably link elongation, it is possible that a node moves from a frequently used part of the input space, for instance near the center of the camera image frame, to a part of the input space that is much less used and hence receives very few learning patterns, if at all. Such node can be considered to never regain its original precision again. Yet, it does still exist and can spoil a trial if the system happens to need feedback from that subspace. In other words, the usability of the relative merge algorithm the way it is implemented suddenly appears to be quite limited as well.

4.4.5 Another wrong way to implement merge

A different approach to detect something in $\mathcal{F}$ has happened, is monitoring the error of different levels in the tree. As long as no change in $\mathcal{F}$ has occurred, one expects to see the average error go down when the tree is descended. After a change in $\mathcal{F}$, one expects the average error to rise when the tree is descended, because the higher levels adapt more quickly to the change in $\mathcal{F}$. However, the latter expectation will never become reality. Instead, the error at higher levels do raise some, but almost never become higher than the error of the root node. The error in the lowest levels of the tree remain almost as low as they already were. It will not raise because the lowest levels will create new nodes that will draw the average error down even before it has the chance to raise much. The mid-levels of the tree however show a considerable rise of the error. Unfortunately, not much can be done with this information, since it can not be decided which nodes should be merged and which nodes should not. It is only helpful for detecting something has happened to $\mathcal{F}$.

4.4.6 An alternative way to implement merge

With detailed insights on the deficiencies of the seemingly simple merge algorithms described above it is clear that a different approach is needed to detect changes in $\mathcal{F}$. One of the main problems of the merge algorithms lie in the fact that it uses information that is very local, while a change in $\mathcal{F}$ affects all nodes in the tree at once. For what the relative merge algorithm is concerned, the performance of only a parent-child pair is measured at some moment in time, using

local information available to the parent and the child, consisting of the error they produce for a common part of some subspace. It does not take advantage of a very important piece of a priori knowledge, which is that when $\mathcal{F}$ changes, *all* nodes will perform worse simultaneously. At the very moment some change in $\mathcal{F}$ occurs, it can safely be assumed that virtually all learning patterns that the system has stored have become worthless because they no longer represent data points in $\mathcal{F}$. These learning patterns will be called *invalid* learning patterns, as opposed to *valid* learning patterns that represent a data point in $\mathcal{F}$. Directly after a change in $\mathcal{F}$, most learning patterns become invalid, while learning patterns created after the change in $\mathcal{F}$ now are valid patterns. Most nodes in the tree will suddenly produce a higher error since they have been trained with invalid learning patterns. The most obvious way to take advantage of this is to monitor the error globally instead of locally. It will take several hundreds of learning patterns before a change in $\mathcal{F}$ can be discovered in order to rule out the possibility that outliers in the error were just coincidences. If the system would be able to pinpoint at what moment some change in $\mathcal{F}$ occurred and also keeps track on when nodes and learning patterns are created, it is possible to discriminate between valid and invalid learning patterns. In order to be able to do this, one will need to sequentially number learning patterns. It will then also be possible to identify perceptrons that have been trained with valid learning patterns only. These perceptrons should become the current perceptron for any node, possibly skipping perceptrons that have also been trained with invalid learning patterns. Nodes that receive very little learning patterns and therefore are unable to adapt to the change in $\mathcal{F}$ should be merged, without regard of the error they produce. Obviously, nodes created after the change in $\mathcal{F}$ should not be merged no matter how little learning patterns they receive, since they have been trained with valid learning patterns only.

The main difference with the approach sketched above with regard to the approaches that have been investigated is that the former uses the fact that all nodes will suffer at the same point in time from a change in $\mathcal{F}$. It is unnecessary to decide if a node should be merged on basis of local information. It even is undesirable since learning patterns are needed to detect a change in $\mathcal{F}$ has occurred, and it will take quite a while before all nodes in the entire tree have received enough patterns to be able to detect that change in $\mathcal{F}$. By using global information in combination with information about when nodes and learning patterns have been created, it should be possible to completely recover from any change in $\mathcal{F}$.

Chapter 5

Performance of Nested approaches

Concerning the performance of the robot system, several different aspects can be regarded. These include precision, computational cost, memory cost and adaptivity. All of these categories will be discussed in this chapter. Also a look will be taken at the limitations of the system that will surface when adaptivity is discussed.

To facilitate a direct comparison between the Nested Perceptron method and the Nested Network method, both are discussed side by side. The implementation of the Nested Network method has undergone some changes that affect its computational cost and memory requirements. These changes will be discussed next.

5.1 Modifications to the Nested Network approach

As was discussed in chapters **??** and **??** the implementations of the tree structure differ. While the Nested Network approach uses virtual nodes to create a path to a leave and its bin, the Nested Perceptron approach does not. The Nested Perceptron approach however has been implemented such that the Nested Network approach can use the same tree structure, making virtual nodes obsolete. This modification only affects memory requirements.

Other modifications have been made concerning the Nested Network approach which reduce the computational cost. When a bin is created, it receives a unique identification consisting of D branch numbers, where D denotes the maximum depth of the tree. That bin will only receive learning patterns belonging to its subspace. For trees with large D, the bins will represent only a very small portion of the total input space. The chance that a particular bin receives a learning pattern decreases dramatically if D increases. For example, bins typically receive only 1.7 learning patterns on the average when $D = 8$. Recall that when some network A is retrained using the Neural Network approach, the most recently added learning pattern of all bins of the subspace represented by A are collected and learned to A. The number of learning patterns collected for the root node consequently is exactly as high as the number of bins in the entire tree. Training the root node with such high numbers of learning patterns would delay the learning process in an unacceptable way. Also, it is not necessary to train a network with such high numbers of learning patterns in order to obtain a good representation of its subspace. Therefore the number of learning patterns that are taught to any node can be reduced to a user-definable number. The learning patterns of a subspace are chosen randomly.

In order to lower the cost of the Nested Network approach even more, networks are only retrained after a trial if the network received at least a user-definable minimum number of new learning patterns. Previously, networks were retrained after a trial even if only one new learning pattern had been received during that trial.

During the testruns these user-definable parameters have been chosen such that they do not influence the precision in a noticeable negative way, at the same time reducing the computational cost as much as possible.

5.2 Parameters influencing system behavior

As has probably become clear, the user can define parameters that influence many aspects of the behavior of the system. Although the general aspects of behavior like tree-building and how learning patterns are handled etc. are fixed, there are about 50 user-definable parameters that have direct influence on the performance of the system. These parameters are listed in section **??**. The reader is encouraged to take a look at that section to better understand the values these parameters have been given when the performance of the system has been measured. It should be noted that it is not trivial to set each of the parameters such that the system shows optimal performance. This can be regarded to be a serious drawback.

5.3 Performance of linear interpolation

In this section a close look will be taken at most of the aspects of performance for the linear interpolation approach.

5.3.1 Results using the input adjustment method

Next, the results obtained using the input adjustment method are discussed. When the input adjustment method is used, $\mathcal{F}$ consists of 5 dimensions. The results of two runs are showed. The first run is depicted in figures **??** and **??**, the second run is depicted in figures **??** and **??**.

Figure **??** shows a run of 50,000 trials that took a total of 72 minutes cpu-time on a Sun SPARCstation 20/51, which amounts to an average speed of 41,400 trials per hour. Figure **??** depicts trial 0 to 3,000 of this run. In order to eliminate peaks in the plots that show the distance of the end-effector and the target position, the data has been averaged. The number of data points that is averaged is shown in the title above the appropriate plots. The number of approximators used is 1,960. This number varies with the value of the split error (see also section **??**). In this run, a split error of $1 \cdot 10^{-4}$ was used. The attainable precision averages to a precision of about 70 μm after 4 feedback steps, which can be considered to be a very good result. The time that is used per trial is virtually linear. For the initial 200 trials the system needs slightly more time per trial than for subsequent trials. This is caused because of the following. The system at least takes 5 feedback steps towards the target position. If however the target position is not within distance ε, subsequent feedback steps towards the target position will be taken. At the beginning of the learning process, the system will often fail to guide the system to at least distance ε within 5 feedback steps. The system will often require all 10 feedback steps for each trial, often without succeeding to get the target position within distance ε. However, when the system's approximation of $\mathcal{F}$ improves, the target position will often be reached in less than 10 feedback steps which saves time, since fewer learning patterns will then be created and processed. This also explains the slight bends in the bins, nodes, memory and patterns plots.

Figure **??** also shows a run of 50,000 trials, but for this run the split error has been set to $1 \cdot 10^{-5}$, a value ten times lower as used in the previously discussed run. This was done in order

to determine whether the precision could be enhanced even more. As can be seen, the number of nodes created (2,427) is somewhat higher, but the precision after 4 feedback steps remains about the same. However, some improvement has been made with respect to the first feedback step, which now comes within 15mm of the target, while the run depicted in **??** has an average distance of about 24mm after one feedback step. On the other hand, the second step shows about equal precision, and third feedback step is even inferior. It is unknown how this particular behavior can be explained. The fourth feedback step shows about the same results as in figure **??**, just as was the case with the second feedback step.

The plot of the number of nodes deserves some special attention. Would one solely look at figure **??** one might be tempted to believe that at some point the number of nodes will reach some upper limit and become constant, or that it will at least gradually raise less quickly. This intuitively seems to be the correct behavior since the system will gradually split all parts of the input space that exceed the split error, until all subspaces have an error less than the split error. Then the number of nodes would not increase any further and the system has learned $\mathcal{F}$ at the desired precision. However, the number of nodes that has been plot in figure **??** shows that it will not become constant within 50,000 trials, it rather starts rising more quickly instead of more slowly after trial 25,000. At first hand this seems unlogical, but this behavior can be explained. First of all, note that this behavior does also occur in figure **??**, but not as extreme. That run only differs with this run in the value of the split error that has been used: its split error is ten times higher. The low split error obviously influences the fluctuations in the growth rate of the number of nodes. A split error of $1 \cdot 10^{-5}$ is so low that most subspaces will be split as soon as enough learning patterns have arrived for these subspaces. Therefore, one should not ask oneself why the rate of growth concerning nodes increases after it descended. The question rather is why the rate of growth decreased in the first place. Imagine a situation in which every subspace in the tree would have equal likelihood to receive a learning pattern, and that the split error is zero, i.e., a split would always occur if enough patterns for a subspace exist. Note that the latter is much like the actual situation, since the low split error will not come into play until the tree has nodes at depth 7. As long as none of the subspaces has received enough patterns to split, the number of nodes will remain 1. After a while, some subspace that received enough learning patterns is the first one to split and create a node at level 1. Other subspaces will now follow soon, since every subspace has equal likelihood to receive a learning pattern and will hence receive an equal number of patterns on the average. The frequency at which new nodes are created will therefore be very low at first, since no or very little subspaces will have received enough learning patterns to initiate a split. Then, many subspaces will be split in a relatively short time. The frequency in which nodes are created will rise to a peak. This frequency will descend as more subspaces have been split. This is because there will fewer and fewer subspaces left at level 1 that have not yet received a node. On the other hand, nodes at level 1 that have received a sufficient number of learning patterns will split and create nodes at level 2. Roughly the same scenario will with the frequency in which nodes are created applies again. At first, few nodes are created at level 2, then the frequency will rise to a peak, to descend again when most subspaces at level 2 have received a node. It is this behavior that causes the growth rate to fluctuate. The rate of growth of the number of nodes in figure **??** decreases at a point where 6 levels in the tree exist, roughly at trial 10,000. It will start rising again at trial 30,000. Shortly before trial 30,000 the first node at level 7 was created. The rise of the growth rate is a consequence of the additional nodes at level 7 that are created. Now take a very close look at the plot of the number of nodes in figure **??**. Each time the growth rate starts increasing after decreasing, a new level in the tree had been created shortly before; level 5 at trial 1,300, level 6 at trial 8,400 and level 7 at trial 40,000. This growth rate ascends less than the growth rate of the number of nodes plot in figure **??**. This is due to the lower split error.

The results of a run with a camera rotation of 90 degrees at trial 1,500 is depicted in figure ??. The system threw away the entire tree at trial 1,504 and started to learn from scratch, pre-learning included. At trial 1,504 the error in the root node exceeded the absolute merge error treshold value (set to 0.15), which initiated the merge of the root node. As one might expect, the precision attained after the first 1,500 trials of this run show great correspondence to the precision attained at the last 1,500 trials. This should come as no surprise since essentially two separate runs are depicted. There is one exception, which is for the memory usage. It might seem as if the system does not free up all memory after a merge. However, the memory used by the system is measured using Unix system calls. Whenever allocated memory is freed, the total amount of memory used by the system will remain equal, since the memory is not actually freed, but reserved for re-use. That is why at trial 1,504 the amount of memory used does not drop to a value near zero. Between trial 1,504 and 1,800 the freed up memory is re-used, which is why the total amount of memory used remains constant for those trials. As soon as no more memory can be re-used, the system will resume to allocate memory. The system has no influence on memory management, which subject to for instance memory fragmentation. It is believed that due to the allocation of many small blocks of memory, fragmentation and overhead become significant factors of memory usage, which is reflected by the memory plot in figure ??. After 3,000 trials, it uses more memory than it would when no rotation would have occurred as in figure ??. The 'time used' plot shows that the system uses slightly more time for a trial after the rotation. The system will often not reach the target within 5 feedback steps when a rotation has occurred, and will therefore often need to take more than 5 feedback steps towards the target. This is simular to the situation at the beginning of the learning process. Consequently, it will take more time on the average before a trial is completed. When the system has learned $\mathcal{F}$ sufficiently well to reach the target within 5 feedback steps on most occasions, the system has regained full speed. The glitch in the 'number of bins' plot is caused by a relative merge that happened approximately at trial 1700. This can be regarded as a fault of the system, since no change in $\mathcal{F}$ has occurred between trial 1500 and 1700, so a merge should not have occurred.

The results of a run with a camera rotation of 30 degrees at trial 1,500 is depicted in figure ??. As can be clearly seen from the distance between the end-effector and the target position, the system fails to recover from this change within 1,500 trials. The system did not throw the entire tree away, but instead used relative merges to get rid of nodes that do not perform well. However, it does not succeed to do this very well. Only a few relative merges occur, but not enough to let the system adapt quickly to the change in $\mathcal{F}$. In section ?? it is explained why relative merging does not work very well.

The results of runs with link elongation at trial 1,500 are depicted in figures ??, ?? and ??. As can be seen from figure ??, an elongation of link 1 with 10 centimeters does not harm the performance very much. The effect of elongating link 1 has the effect of lifting the robot with 10 centimeter. The irregularities shown in the '3rd step' and '4th step' plots are caused by peaks. Since the plots are averaged using an average filter of 50 points, the peaks are spread out over 50 neighbouring points. Apparently, the elongation of this link can be handled by the system very well. An alternative change in $\mathcal{F}$ forced by elongation of link 2 of 10 centimeters on the other hand is not handled very well. Although the precision after 1 and 2 feedback steps recovers reasonably, the performance of steps 3 and 4 is crippled and the system is not able to adapt within 1,500 trials. Note that no relative merge occurred, even though it seems very appropriate. This stresses the weakness of the relative merge algorithm. In contrast, if link 2 is shortened with 10 centimeters instead of elongated, the system is able to recover almost entirely. In this run, the relative merge seems to do something useful after all. It is not known why a relative merge happens here but not in the previously discussed run. It could very well be a matter of coincidence.

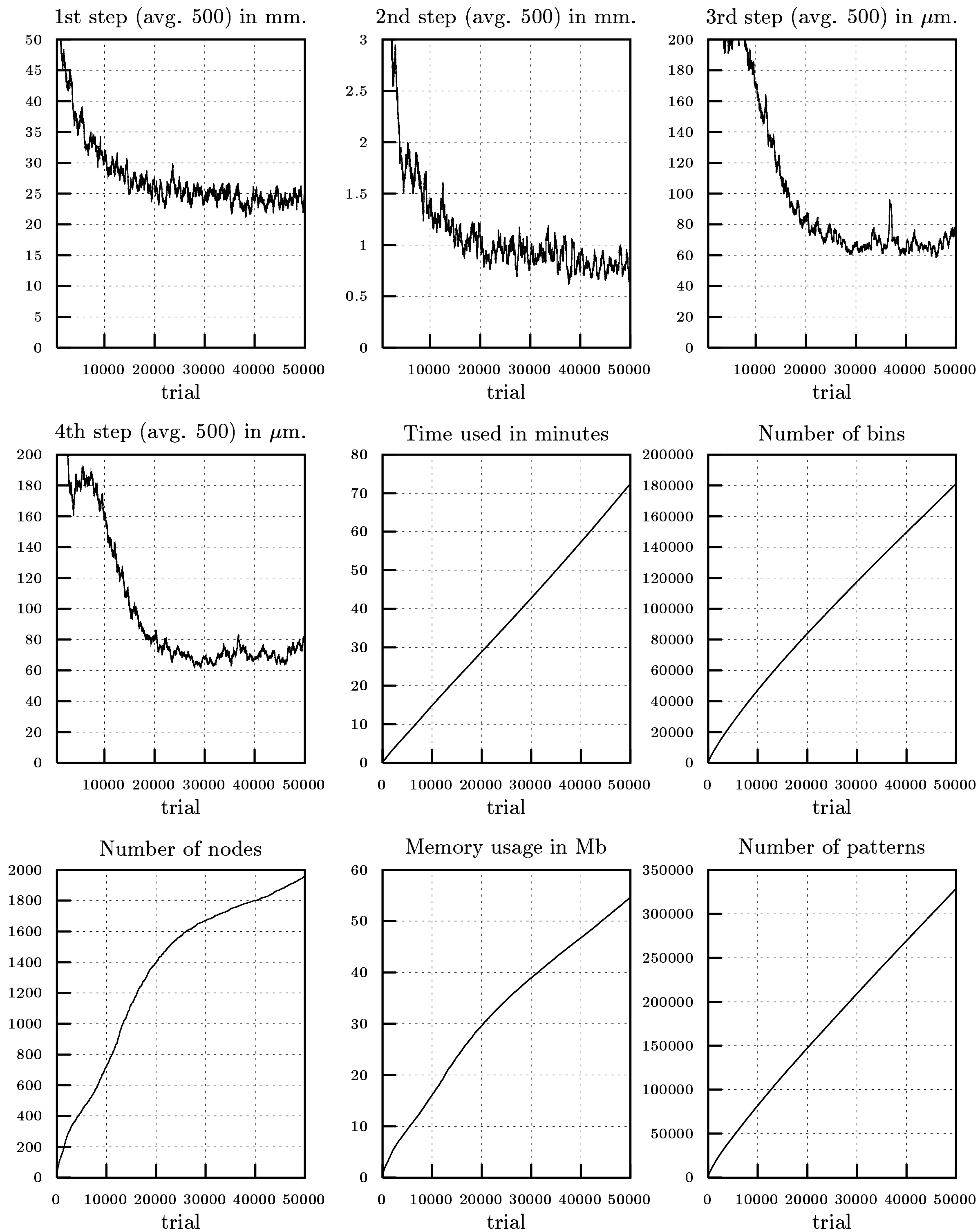


Figure 5.1: Results of learning $\mathcal{F}$ using the input adjustment method and a split error of $1 \cdot 10^{-4}$

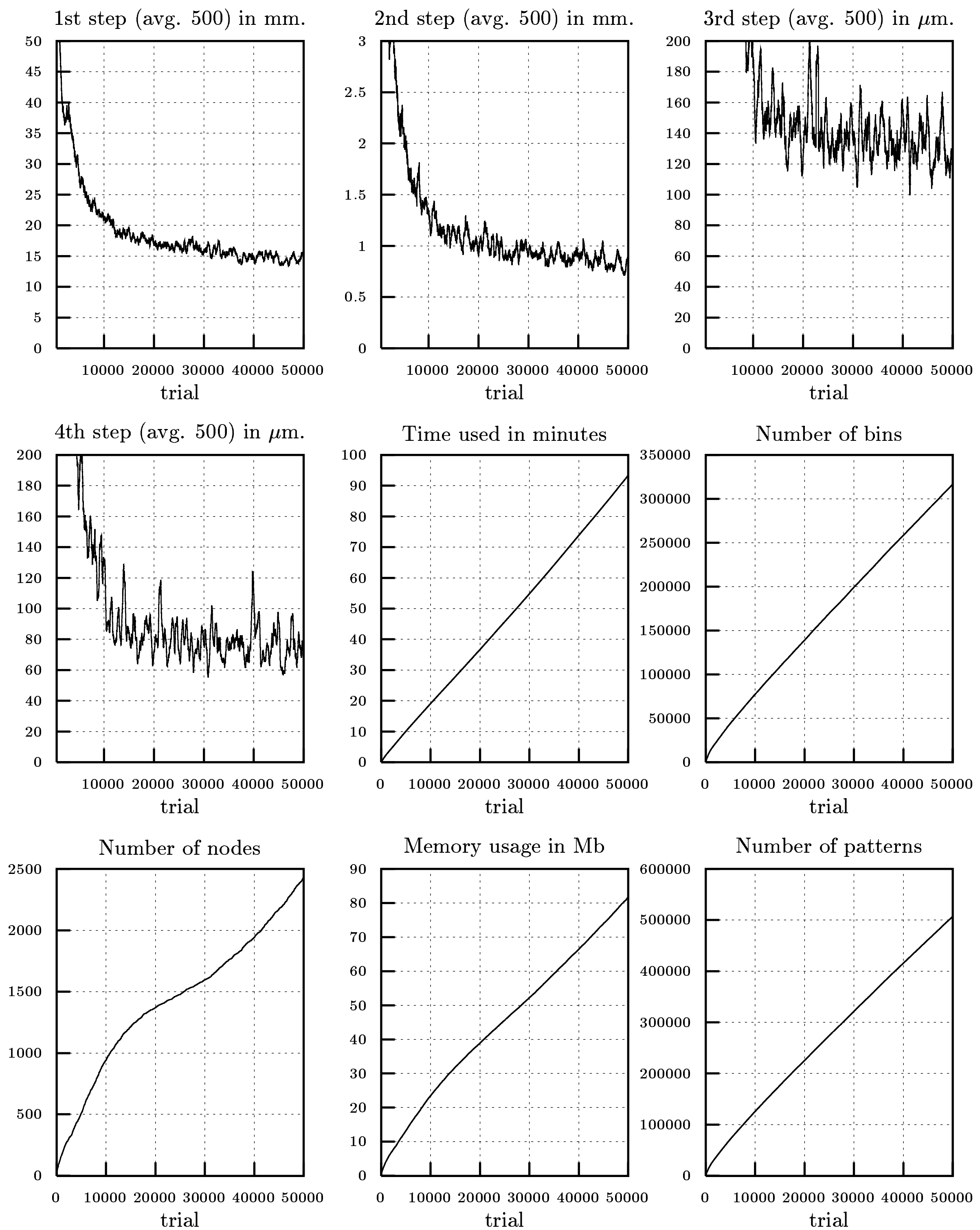


Figure 5.2: Results of learning $\mathcal{F}$ using the input adjustment method with split error $1 \cdot 10^{-5}$

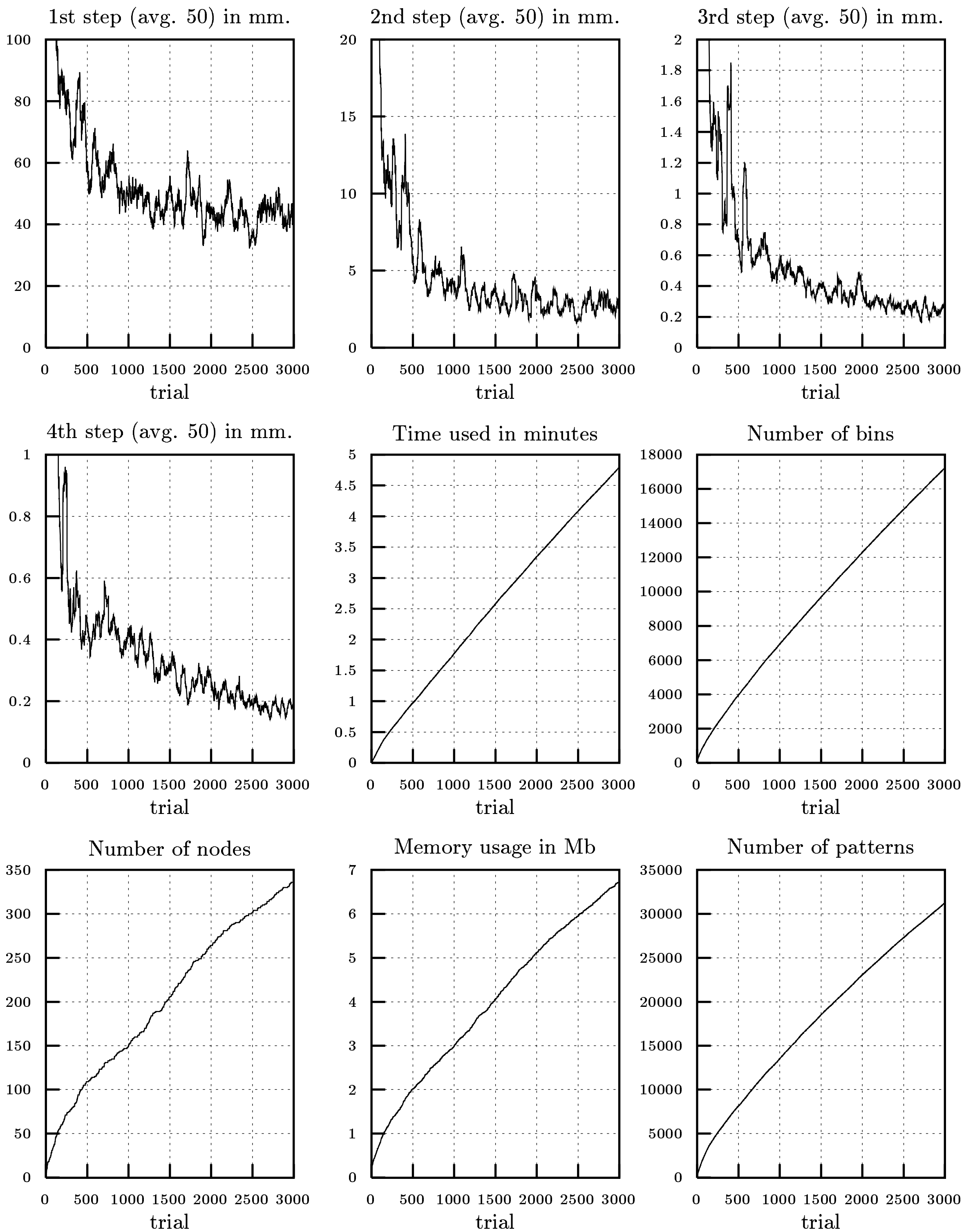


Figure 5.3: Enlargement of figure ??. Results of learning $\mathcal{F}$ using the input adjustment method and split error $1 \cdot 10^{-4}$

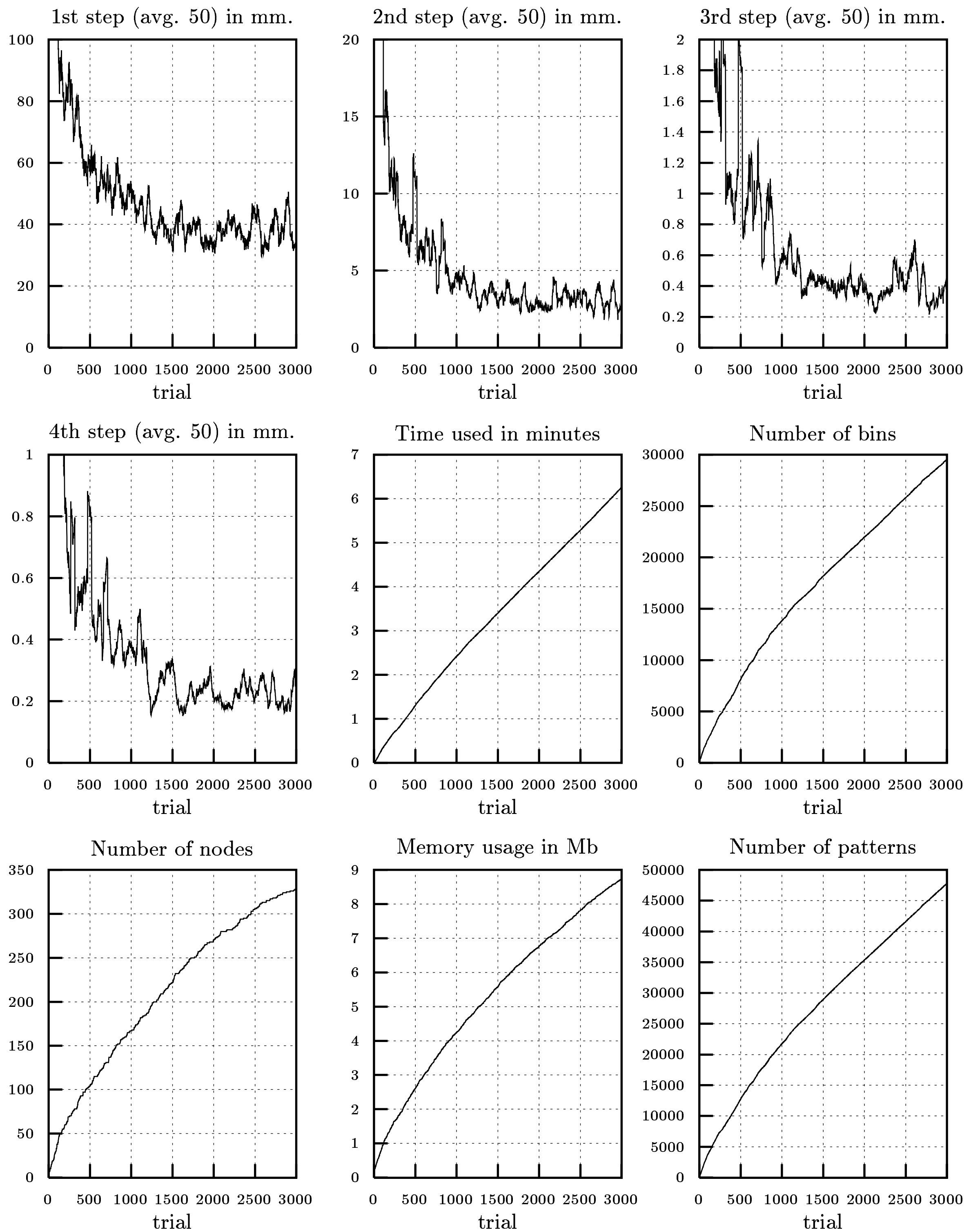


Figure 5.4: Enlargement of figure ??. Results of learning $\mathcal{F}$ using the input adjustment method and a split error of $1 \cdot 10^{-5}$.

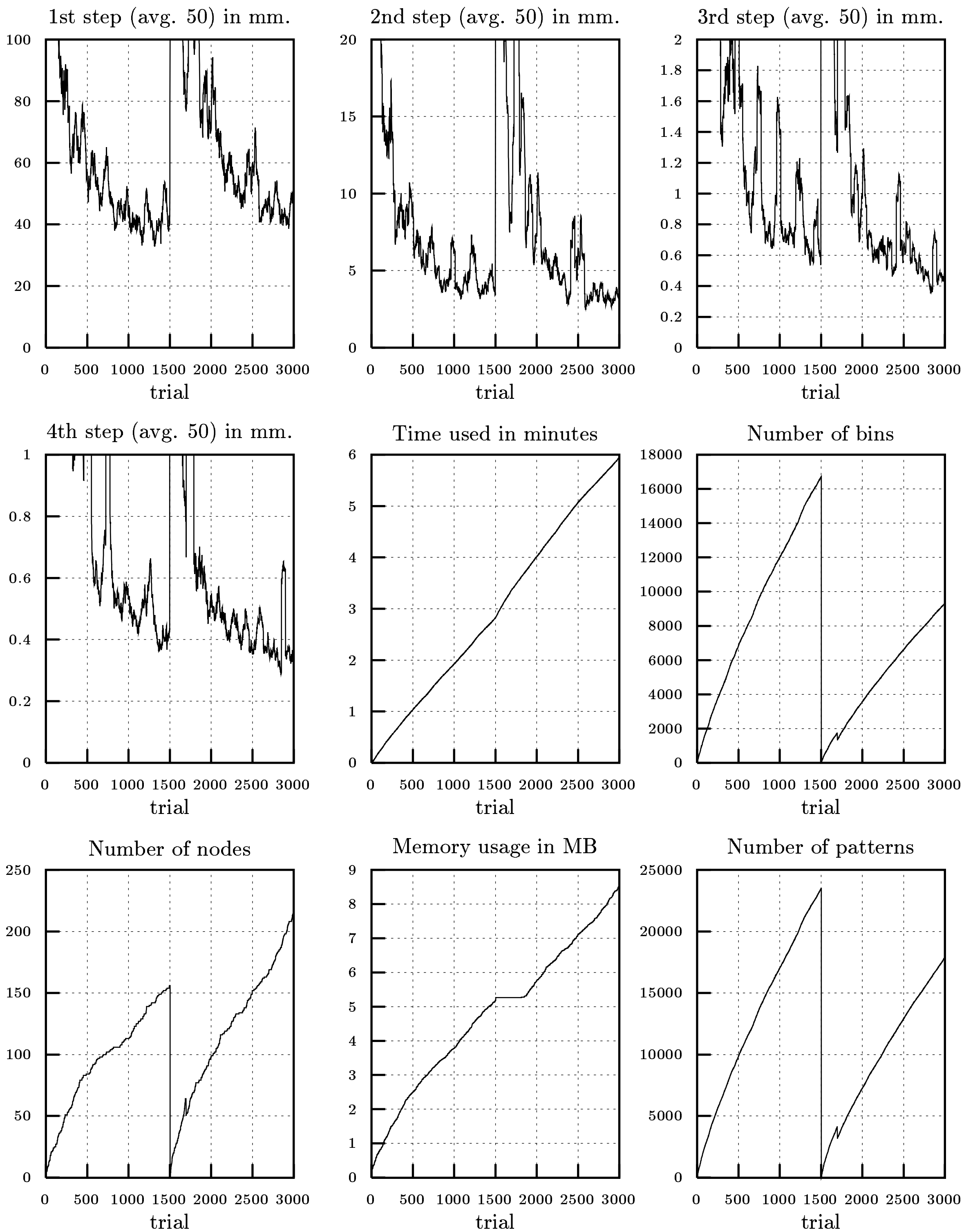


Figure 5.5: Results of learning $\mathcal{F}$ using the input adjustment method; Camera rotation of 90 degrees at trial 1,500 using a split error of $1 \cdot 10^{-4}$

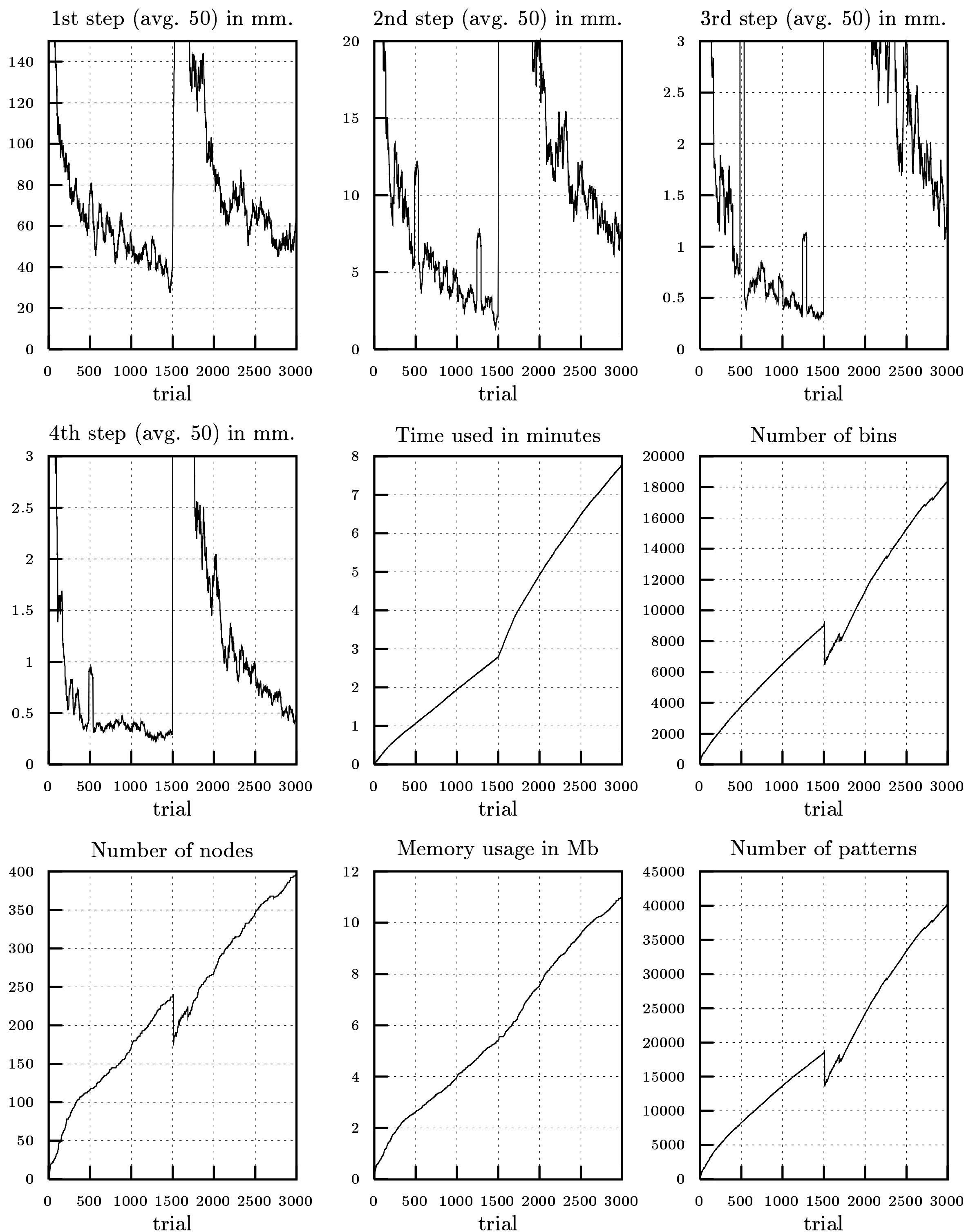


Figure 5.6: Results of learning $\mathcal{F}$ using the input adjustment method; Camera rotation of 30 degrees at trial 1,500 using a split error of $1 \cdot 10^{-4}$

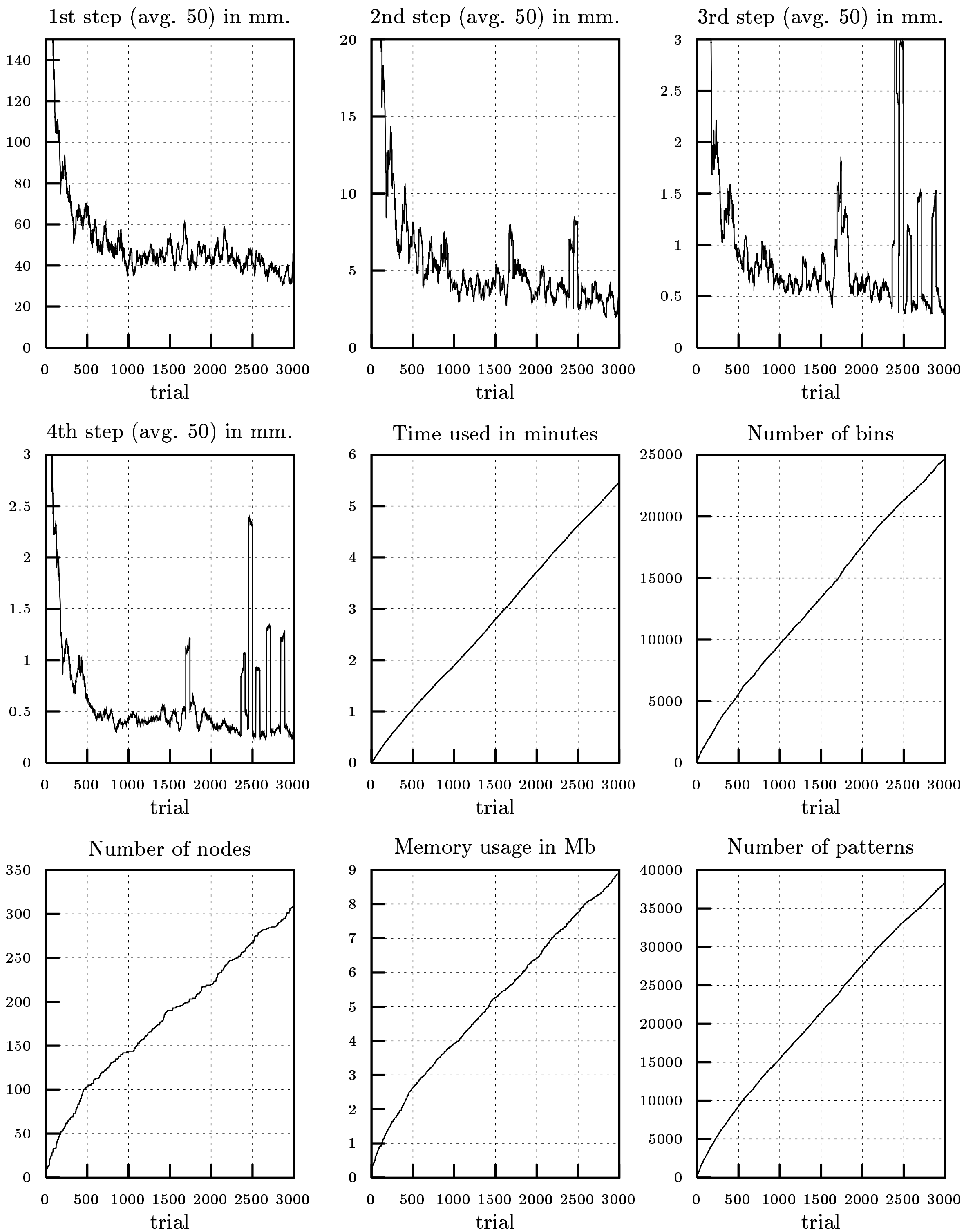


Figure 5.7: Results of learning $\mathcal{F}$ using the input adjustment method; Elongation of link 1 with 10 centimeters at trial 1,500 using a split error of $1 \cdot 10^{-4}$

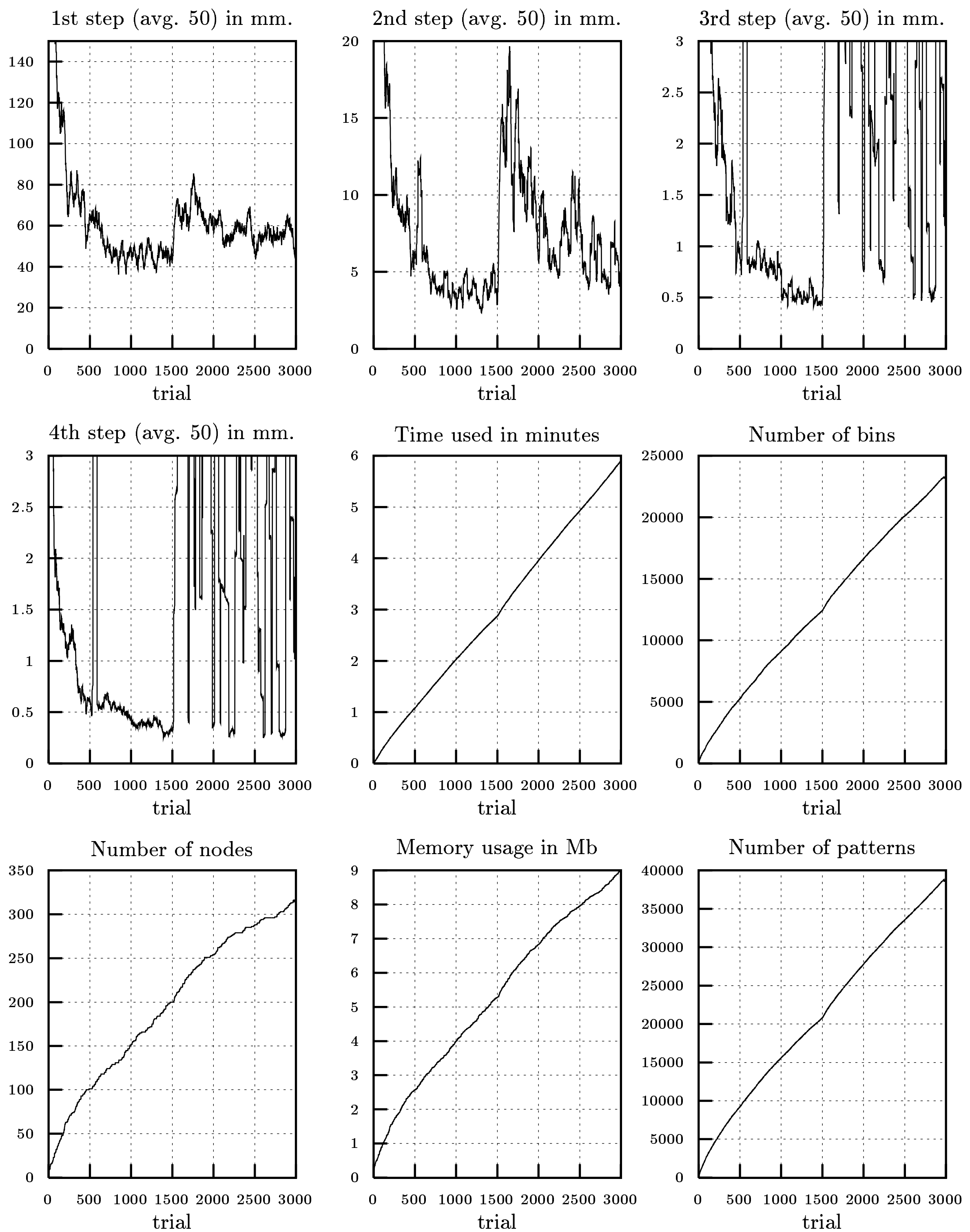


Figure 5.8: Results of learning $\mathcal{F}$ using the input adjustment method; Elongation of link 2 with 10 centimeters at trial 1,500 using a split error of $1 \cdot 10^{-4}$

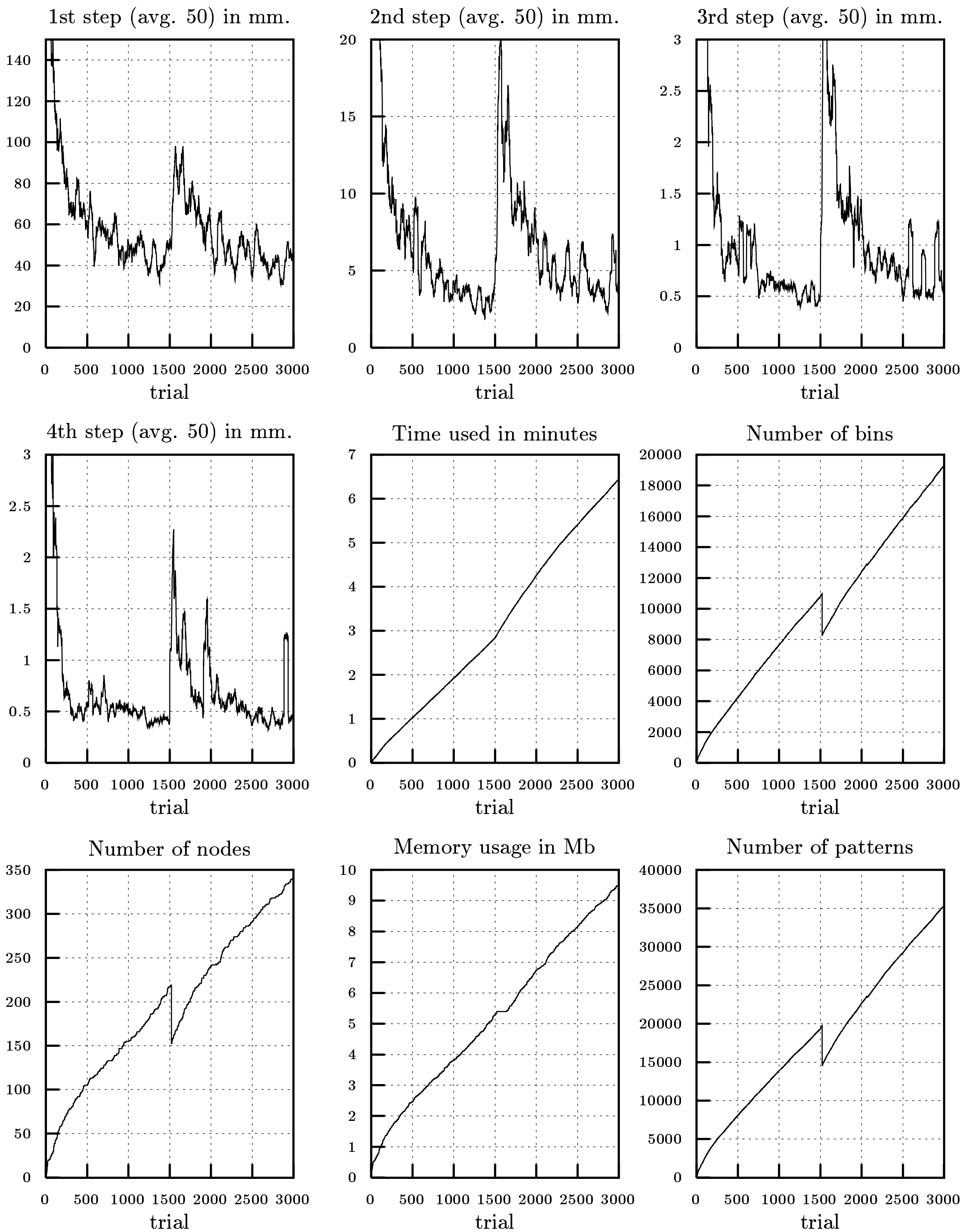


Figure 5.9: Results of learning $\mathcal{F}$ using the input adjustment method; Elongation of link 2 with -10 centimeters at trial 1,500 using a split error of $1 \cdot 10^{-4}$

5.3.2 Results without using the input adjustment method

All previously described runs were also done without making use of the input adjustment method. In these runs, $\mathcal{F}$ has a dimensionality of 8 and is therefore harder to approximate. This becomes immediately clear when figure **??** is compared to figure **??**. The high precision that could be attained with help of the input adjustment method is not attained here. However, when the split error is lowered to $1 \cdot 10^{-5}$ a comparable precision is reached. Apparently, a split error of $1 \cdot 10^{-4}$ prevents the system from performing well. The important differences between figure **??** and **??** can be found in the 'time used', 'number of bins', 'number of nodes', 'memory usage' and 'number of patterns' plots. Since the system now has to deal with an $\mathcal{F}$ of increased dimensionality, the time used and the number of nodes that are created are higher. The number of bins and number of learning patterns stored are however lower. The difference between these numbers are caused by the fact that when the dimensionality of $\mathcal{F}$ equals 8, there will exist a class of bins and learning patterns that only represent subspaces in higher levels of the tree, but can never reach deeper levels in the tree. Since only one bin for each node is used to store them, those learning patterns will often be overwritten. See also section **??**. The increased number of nodes and the decreased number of stored learning patterns and bins apparently compensate each other for what memory usage is concerned, since the 'memory usage' plots of **??** and **??** do not differ very much.

The runs depicting camera rotation and link elongation show striking similarity with the same runs but using the input adjustment method. It is clear that the mechanisms of relative merging and the adaptive behavior of the system is hardly influenced by the dimensionality of $\mathcal{F}$.

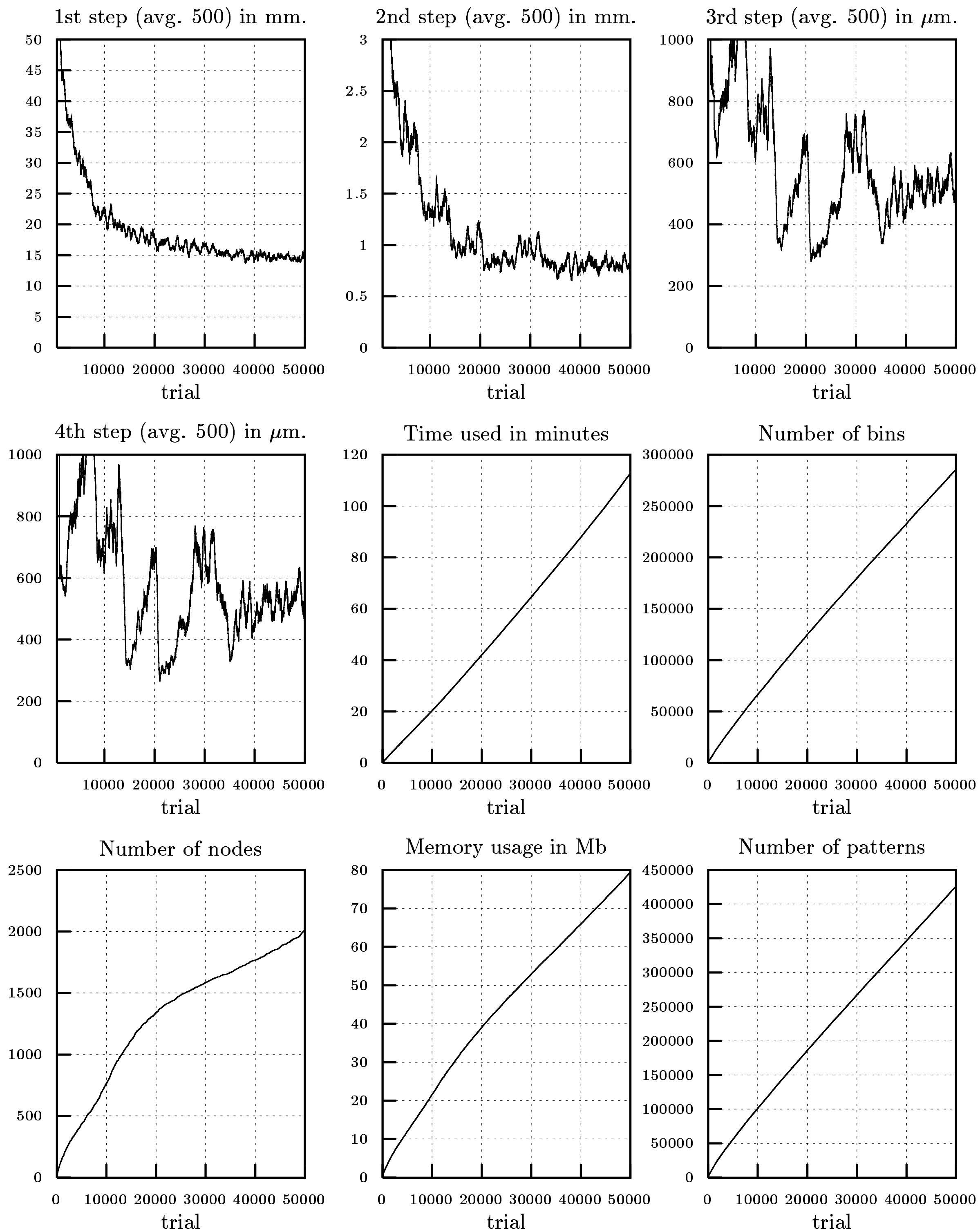


Figure 5.10: Results of learning $\mathcal{F}$ without using the input adjustment method using a split error of $1 \cdot 10^{-4}$.

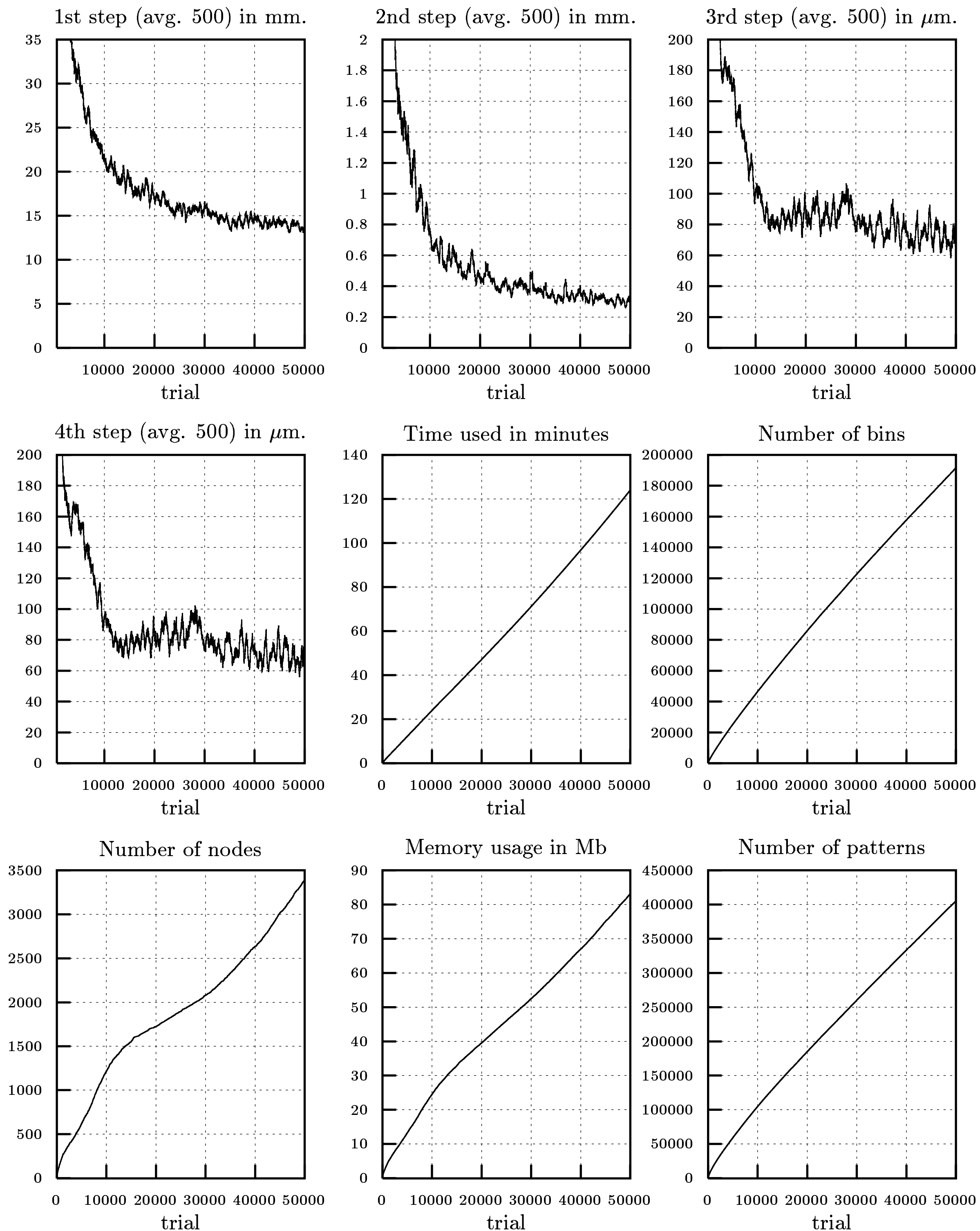


Figure 5.11: Results of learning $\mathcal{F}$ without using the input adjustment method using a split error of $1 \cdot 10^{-5}$

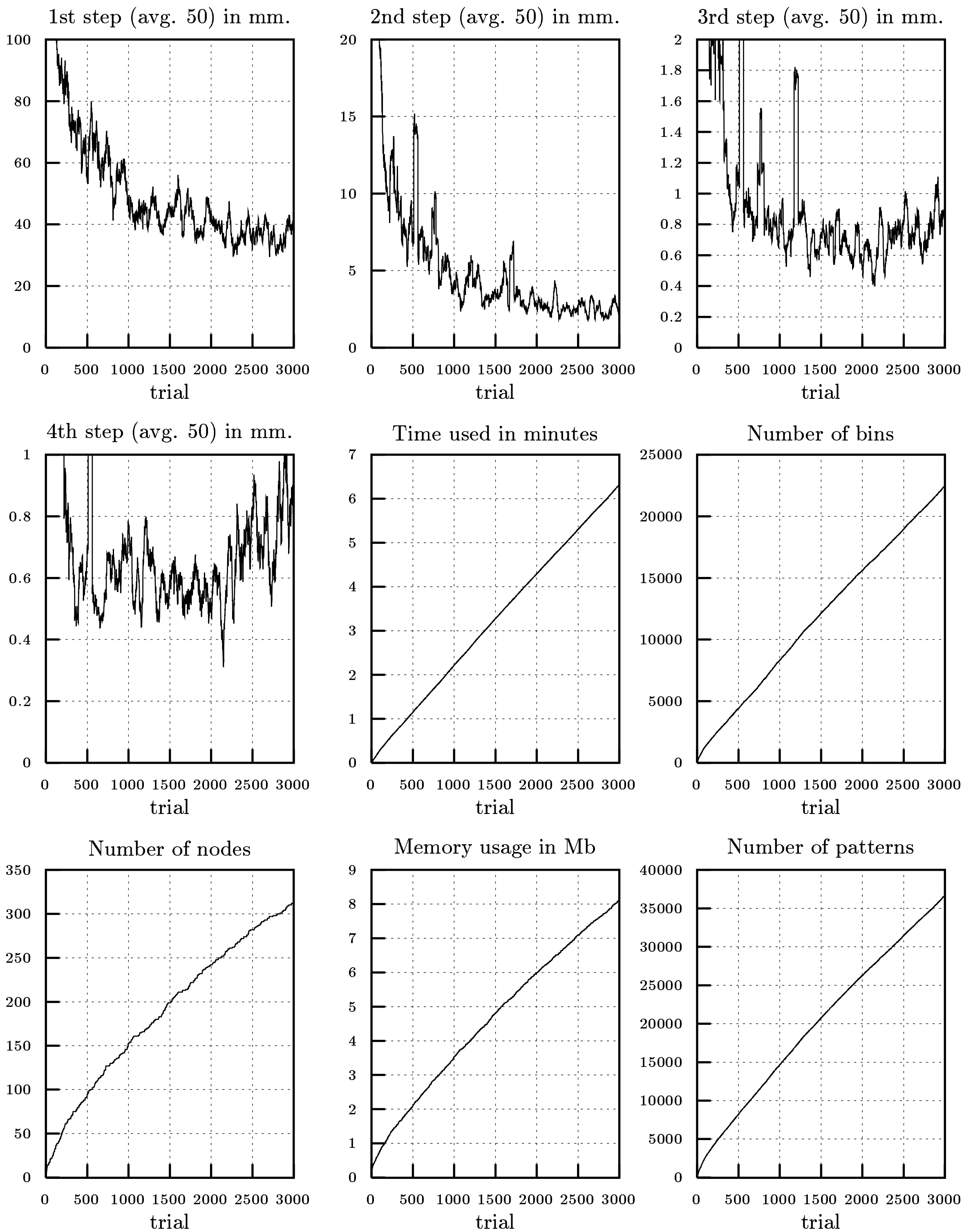


Figure 5.12: Enlargement of figure ??. Results of learning $\mathcal{F}$ without using the input adjustment method and a split error of $1 \cdot 10^{-4}$

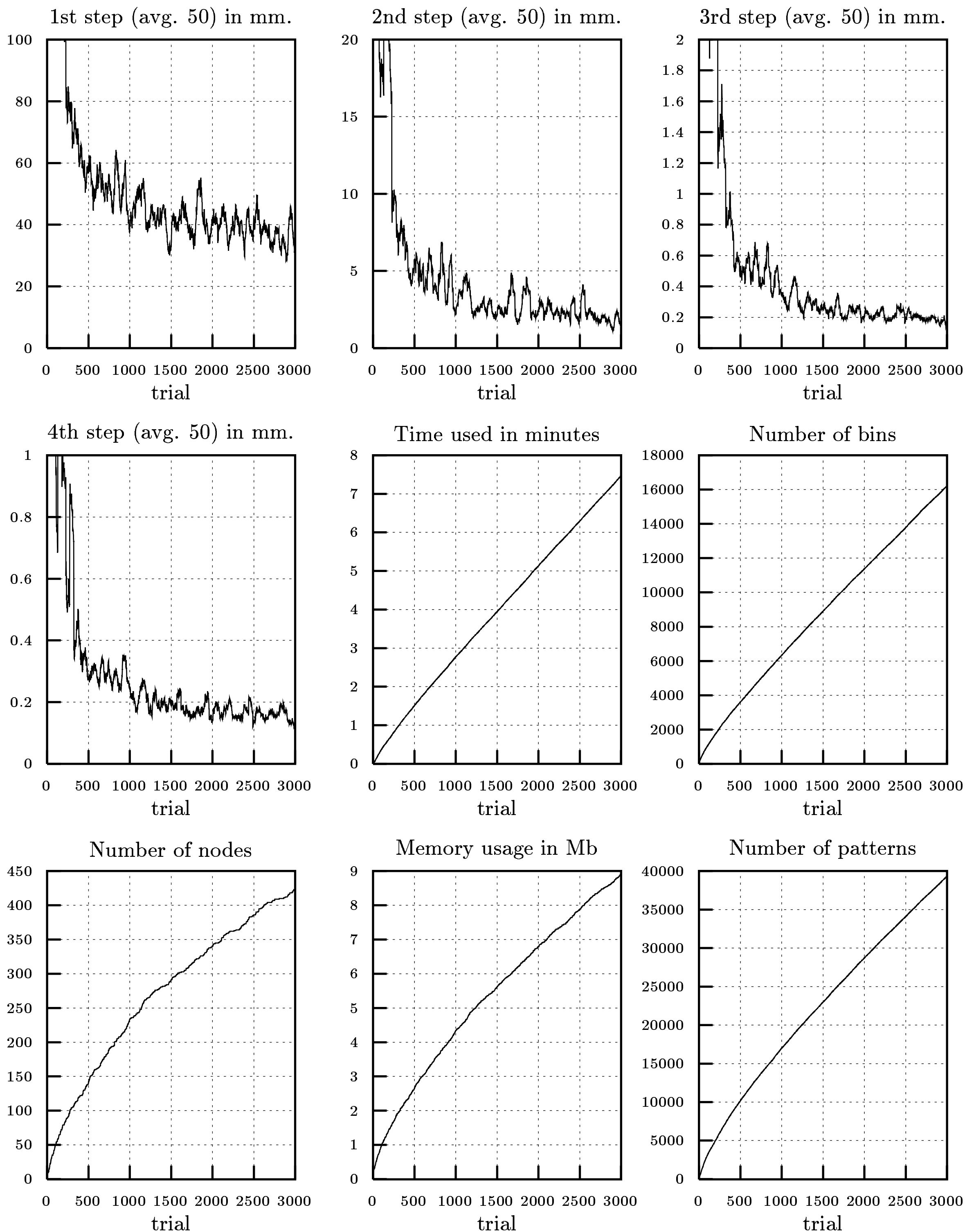


Figure 5.13: Enlargement of figure ??. Results of learning $\mathcal{F}$ without using the input adjustment method and a split error of $1 \cdot 10^{-5}$

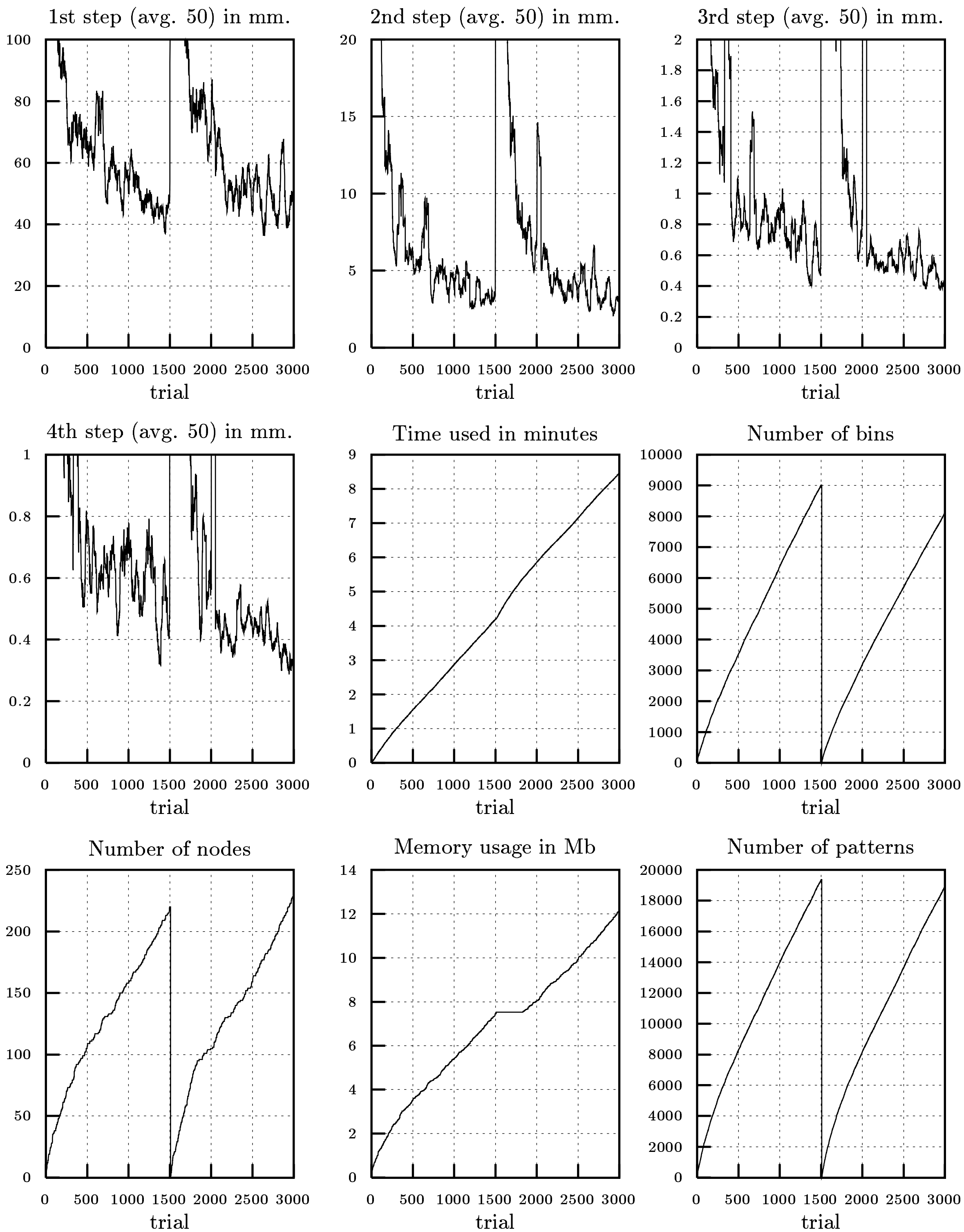


Figure 5.14: Results of learning $\mathcal{F}$ without using the input adjustment method; Camera rotation of 90 degrees at trial 1,500 using a split error of $1 \cdot 10^{-4}$

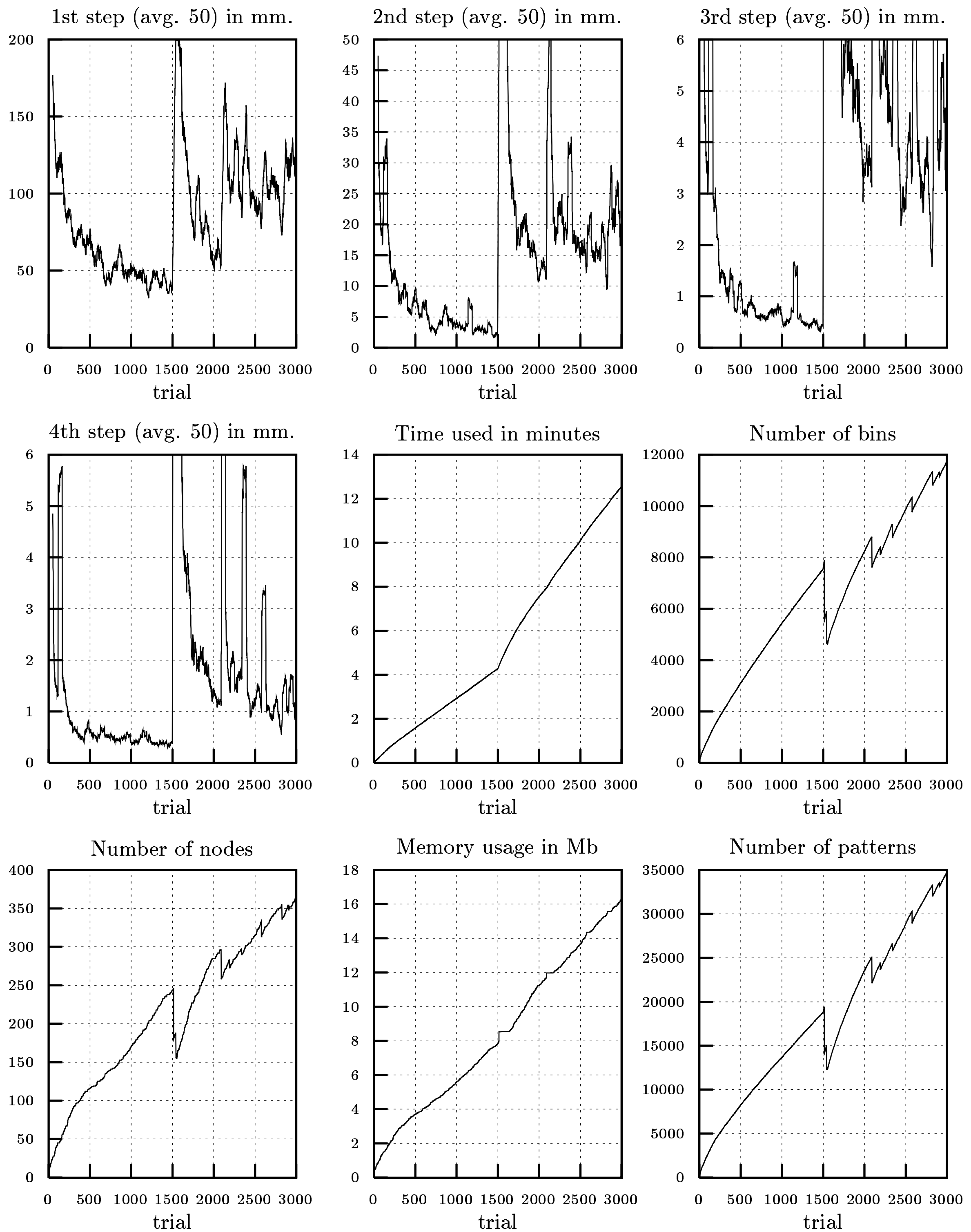


Figure 5.15: Results of learning $\mathcal{F}$ without using the input adjustment method; Camera rotation of 30 degrees at trial 1,500 using a split error of $1 \cdot 10^{-4}$

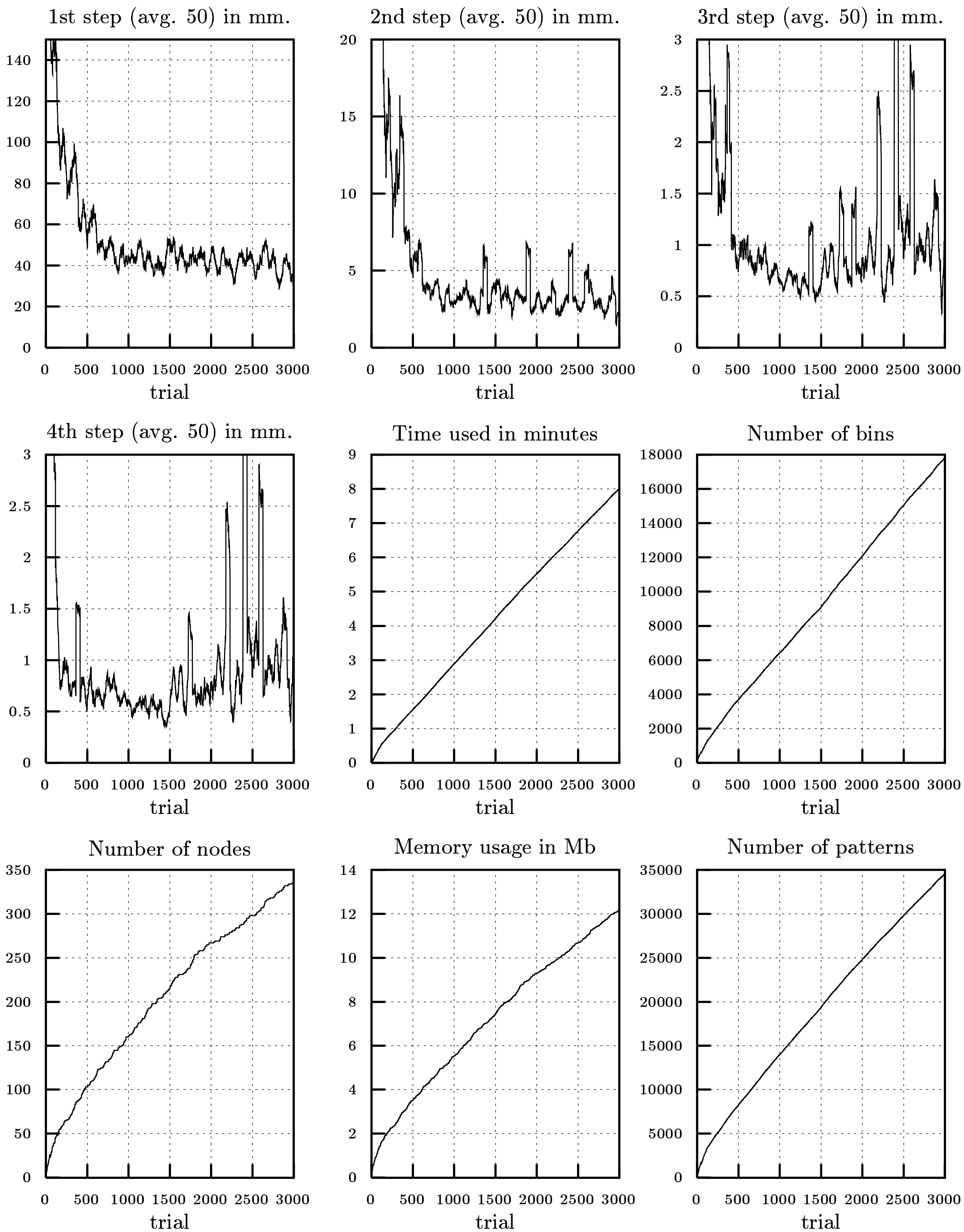


Figure 5.16: Results of learning $\mathcal{F}$ without using the input adjustment method; Elongation of link 1 with 10 centimeters at trial 1,500 using a split error of $1 \cdot 10^{-4}$

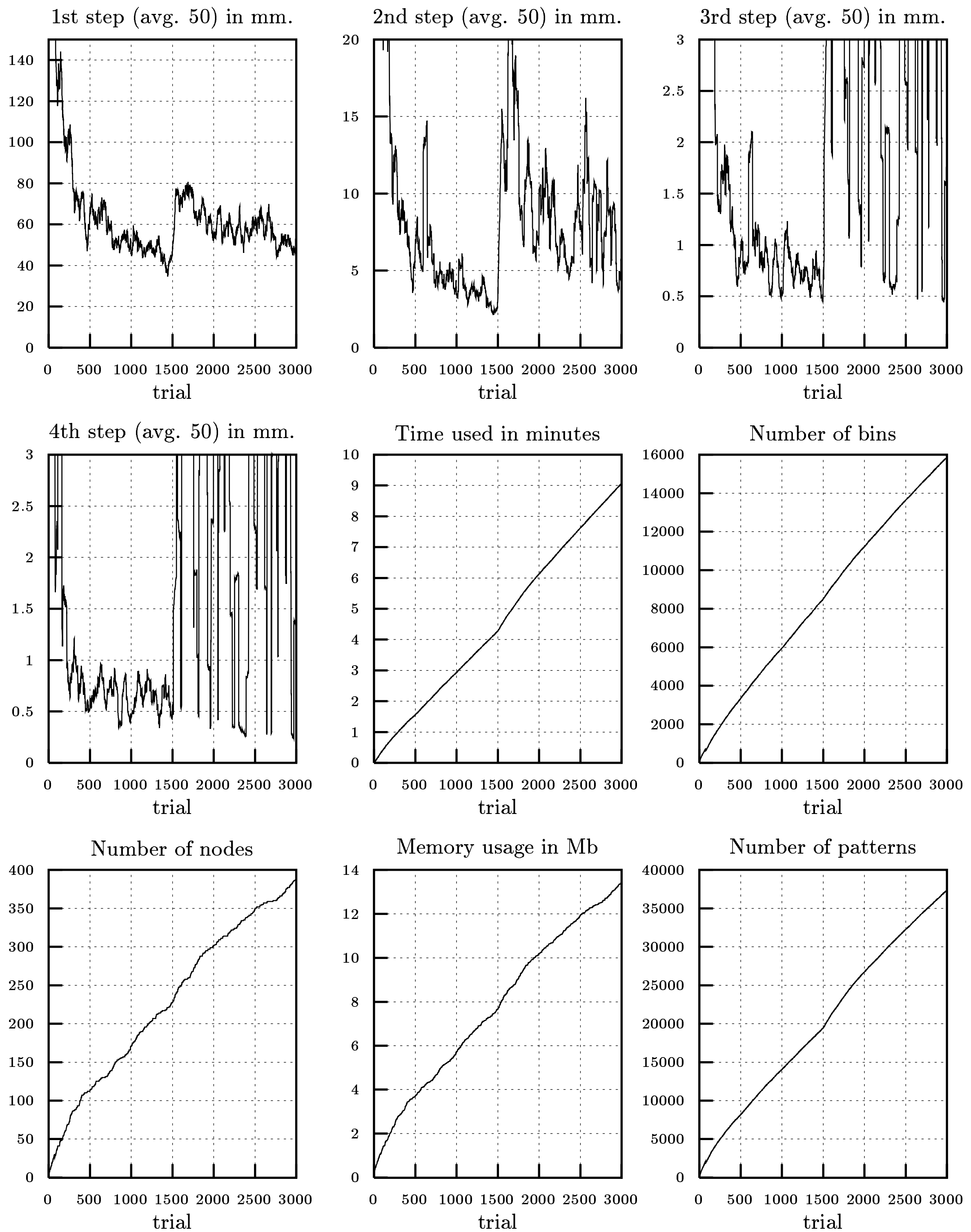


Figure 5.17: Results of learning $\mathcal{F}$ without using the input adjustment method; Elongation of link 2 with 10 centimeters at trial 1,500 using a split error of $1 \cdot 10^{-4}$

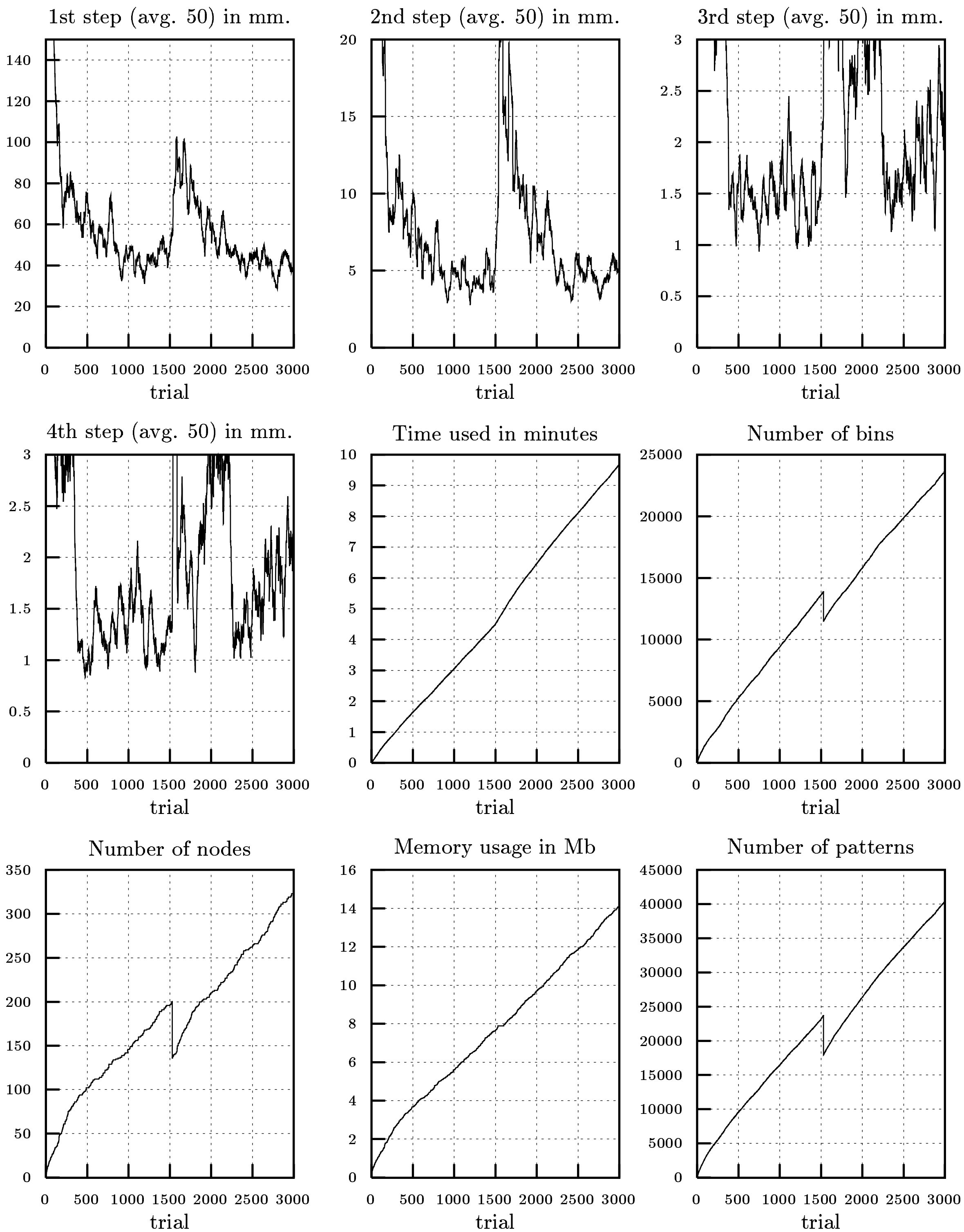


Figure 5.18: Results of learning $\mathcal{F}$ without using the input adjustment method; Elongation of link 2 with -10 centimeters at trial 1,500 using a split error of $1 \cdot 10^{-4}$

Chapter 6

Conclusion

In this thesis several approaches to solve a high-dimensional eye-hand mapping function have been researched and compared to each other. All approaches were tested in a simulated robot environment running on Sun SPARCstations. The robot simulator simulated an OSCAR-6 robot having 6 degrees of freedom. After fixating and restricting movement of several joints, three degrees of freedom were left and used in the experiments, sufficient for performing most pick-and-place operations. The robot has been equipped with joint sensors and a camera. While learning, the system creates learning patterns which are correct points in the eye-hand mapping function. The system attempts to build a representation of the function solely by utilizing these learning patterns. The goaled minimum distance between the end-effector and the target position was set to half a millimeter.

Early approaches using Kohonen networks, feed-forward networks, as well the Nested Network method using a high-dimensional tree of feed-forward networks were compared to the Nested Perceptron method, which builds a high-dimensional tree containing perceptrons. Precision, speed, adaptivity as well as memory usage were taken into account when measuring the performance of these methods.

Using a single feed-forward network with 25 hidden units, the goaled precision of one millimeter could not be attained. It appeared that the eye-hand mapping function was too complex to be approximated with a single feed-forward network. Raising the number of hidden units to 45 did not show any positive effects. A $7 \times 7 \times 7 \times 7 \times 7$ Kohonen network needed about 10,000 trials to reach maximum precision, which was not anywhere near to the goaled one millimeter either, although the results were better than when a single feed-forward network was used. However, the Kohonen networks required considerable computational resources and many trials to raise the maximum precision. Raising the number of networks is therefore regarded unfeasible, since the computational cost would become very large.

The performance of the Nested Perceptron method is very well for a non-changing eye-hand mapping function. The computational resources are thousands of times lower as when using nested feed-forward networks, while at the same time the precision is extreme: the attainable precision is a distance of only 70 micron on the average between the target and the end-effector after 3 feedback steps. A speed of 11 trials per second can be maintained on a Sun SPARCstation 20, which is very fast compared to the Nested Network method. In contrast, a run of 2,000 trials using nested feed-forward networks in a tree structure took about 50 hours and just barely managed to approximate the target within one millimeter after 3 feedback steps.

On the area of adaptivity, the results are not very encouraging. The suggested merge algorithms suffer from theoretical deficiencies. Using multiple perceptrons to enhance adaptivity will not increase the adaptivity of the system to an acceptable level. Adapting to a minor change in the

eye-hand mapping function when depending solely on the adaptive nature of multiple perceptrons takes more trials than it would have when the function was to be learned from scratch. However, they were invented to be able to adapt to changes in the eye-hand mapping function quickly and without having to relearn it from scratch. A different merge algorithm has been proposed that may solve these deficiencies.

In general, one could say that the Nested Perceptron method beats every other method known thus far when it comes to computational cost and attainable precision. However, the adaptivity of Nested Perceptron is unsatisfying, despite the usage of the split and merge algorithm. The local approach of the merge algorithms suffer from great deficiencies that have shown to be unsolvable. They relay mainly on the faulty premise that it is trivial to decide if a child node performs better as its parent node. As has been made clear, this is not trivial at all. Subsequent approaches in dynamically merging subtrees should focus on the detection of an change in $\mathcal{F}$ and have the ability to discriminate between valid and invalid learning patterns. Although this approach has not yet been implemented, it seems a very sensible approach to solve the problem of lack of adaptivity that the researched merge algorithms suffer from.

Appendix A

Internals of the Software

The system consists of several programs, which all have been written in the C programming language under a Unix-environment:

- A low-level program which controls the robot, retrieves camera information and reads joint angles,
- The robot simulator,
- The logic that learns the eye-hand mapping function $\mathcal{F}$.

In this appendix a closer look will be taken at the latter part of the software, which effectually consists of the tree data structure as well as the Nested Network approach and the Nested Perceptron approach for solving the eye-hand mapping function $\mathcal{F}$.

A.1 The data files

In order to be able to analyze a run of the system, several measurements are made and stored in files. The system expects a `data` directory in the current directory. There are three methods that can be selected for a run, which are the neural network approach, the linear interpolation approach using input adjustment, and the linear interpolation approach without using the input adjustment approach. Depending on which of the three methods has been selected, the data will end up in respectively `data/nnw`, `data/lin5` or `data/lin8`. The `5` and `8` refer to the number of inputs that the method uses. The directories are created if they do not yet exist. Since the user often will run the system multiple times while experimenting with several values in the config file or testing the system itself, the aforementioned directories contain directories itself, which are simply numbered `0` to `9` and `a` to `z`. Each directory will contain one run, and the directory will be created if it does not yet exist. Data will end up in a directory that does not yet exist. If all directories have been used (36 of them for each selected method), then it is time to delete or rename some of them to make room for new directories.

Each run generates the following 11 plain text files:

Filename	Description
1step	Distance to target position after 1 feedback step
2step	Distance to target position after 2 feedback steps
3step	Distance to target position after 3 feedback steps
4step	Distance to target position after 4 feedback steps
5step	Distance to target position after 5 feedback steps
nodes	Number of nodes in the tree
bins	Number of bins in the tree
mem	Memory used in kilobytes
patterns	Number of learning patterns stored
abs_err	The error for each level in the tree
info	General information about the run

Except for `info`, `abs_err` and `pat_file`, all files have the trial number in their first column, the relative data at the second column, and the passed system time used measured in minutes in the third column. The `abs_err` file has the trial number in its first column, and the error of level n in column $n+1$. If no exists for a particular level, the column will be empty.

The `info` file contains all sorts of information that can be used to analyse general aspects of the run, most notably the values of all parameters as defined in the config file, average number of trials per hour, the tree structure shown for each level after learning, and the name of the host the system ran on.

In addition to these files, one will also find a `pat_file`. If learning patterns are stored to disk, the patterns will be stored in this file. Otherwise, the file will remain empty. The neural network approach also creates a `weight_file`, in which the weights of the feedback networks are stored. Both `pat_file` and `weight_file` contain data that is not suited to analyze. The data herein is not stored as plain text.

A.2 The configuration file

The config file contains all user-definable parameters. Prior to a run of the system, the user can alter any parameter. The order of the entries of in file can be arbitrary. Comment starts with a '`#`' symbol, blank lines are allowed. If any parameter has been multiply defined or not at all, or if an unrecognized parameter has been encountered, or if a floating point value has been assigned to an integer, the system will halt and output an appropriate error message and a line number of the config file where it found the error. All entries within the config file will end up in the `info` file mentioned in the previous section. An example config file is shown below.

```
# The configuration file for the robot.
# Not all parameters are used by all three methods. The three methods
# are:
#
#    lin5       5 inputs, using input adjustment method
#    lin8       8 inputs, without using input adjustment method
#    nnw        5 inputs, using feed-forward networks
#
# For each parameter it will be shown what methods are affected by it.
# The order of the entries can be arbitrary.

# First follow the boundaries for the joints and the camera.
# [boundaries 0-4: lin5 lin8 nnw]  [boundaries 5-7: lin8]
```

```
boundary_0_low        -120.0
boundary_0_high          0.0
boundary_1_low          65.0
boundary_1_high        180.0
boundary_2_low        -200.0
boundary_2_high        100.0
boundary_3_low        -200.0
boundary_3_high        100.0
boundary_4_low        -200.0
boundary_4_high        100.0
boundary_5_low        -100.0
boundary_5_high        100.0
boundary_6_low        -100.0
boundary_6_high        100.0
boundary_7_low        -100.0
boundary_7_high        100.0

# The minimal distance between the end-effector and the target object.
# If the distance becomes less no further feedback steps are taken.
# [lin5 lin8 nnw]
min_target_distance       0.05

# Unknown [probably nnw]
split_extension           2.0

# Relative merge error. If a higher node produces a better result then a
# lower node in the tree with a difference of at least this value, the
# lower node and its subtree are merged. To date, it has been found
# that relative merging does more bad than good. [lin5 lin8]
merge_error             99.05

# If the error in a subregion of a node is higher than this value, and a
# certain amount of patterns exist for that node, the node is split. The
# subregion gets its own node. [lin5 lin8 nnw]
split_error               0.0001

# At most this number of steps towards the target are taken. [lin5 lin8 nnw]
feedback_steps          10

# The tree will not be split to depths beyond this value. [lin5 lin8 nnw]
tree_depth                8

# Rotate the camera at this trial. [lin5 lin8 nnw]
rotate_trial          1500

# Rotate the camera this number of degrees at trial rotate_trial.
# [nnw lin5 lin8]
rotate_degrees          90.0
```

```
# The number of matrices for each node. Higher values consume significantly
# more memory. [lin5 lin8]
nr_of_matrices           5

# The maximum number of patterns in a matrix. If this number has been reached,
# the next matrix will be used. nr_of_matrices / max_pats_in_mat are discarded.
# [lin5 lin8]
max_pats_in_mat        1000

# At least this number of patterns have to be present for a subarea before
# it is split. Do not enter a value below 25 here. This avoids peaks that
# are caused by nodes generating a movement while being trained with too
# few patterns. [lin5 lin8 nnw]
min_pats_for_split 50

# At least this amount of movement must be made by the robot if it is to
# be regarded as an actual move. This accounts for each joint separately.
# [lin5 lin8 nnw]
minimal_joint_move 0.01

# During learning, the error at each depth in the tree is monitored.
# This is done to detect if the error suddenly raises. It averages
# the error over monitor_ave number of patterns. [lin5 lin8]
monitor_ave            100

# If the error of a node becomes greater than this value on the average
# calculate over am_error_iters points (see below) an absolute merge is
# performed on that node. [lin5 lin8]
am_error_tresh 5.0

# At least this number of patterns have to average above am_error_tresh
# before an absolute merge is performed on the node. [lin5 lin8]
am_error_iters 50

# This parameter is only used if the input adjustment method is disabled.
# In that case, the input space has 3 additional dimensions and 8 times
# as much partitions in theory. However, only 1 of the additional 7 * 32
# partitions is needed: the one that intersects with
#
#             (th1, th2, x, y, z, 0, 0, 0)
#
# This way the number of partitions still remains 32, even for 8 inputs.
# However, if the tree becomes deeper, the boundaries around the last
# 3 zeroes are narrowed. They are each multiplied by the factor below.
# That way, deeper nodes will approximate a smaller area in the extra
# 3 dimensions, which enhances the performance. [lin8]
error_area_shrink 0.20

# Learning patterns can be either stored to a file or to memory. Storing
```

```
# patterns in memory requires large amounts of memory, it however speeds
# up the needed run-time considerably for the Nested Perceptron approach.
# It does not for the Nested Network approach since that method spends
# nearly 100% of its time training patterns to nodes. Be aware that large
# runs (> 10000 trials) can cause memory problems. For extra-ordinary long
# runs it is therefore recommended to save patterns to a file.
# 0 = store patterns to file, anything else = store patterns in memory.
# [nnw lin5 lin8]
use_pat_file            0

# During a run, information about how the tree is build up will be printed
# after every depth_info trials. [nnw lin5 lin8]
depth_info              5

# Print all kinds of info during a run (1), or don't (0) [nnw lin5 lin8]
verbose_output          1

# The speed of the robot is defined here. If a slow speed is selected (like
# 0.1) the robot will move slower to the target position and the system will
# have the time to create learning patterns during the movement of the robot.
# This results in more learning patterns that are created and could enhance
# accuracy. Also useful if the robot easily hits the boundaries of the reach
# space. If the robot moves very fast, and a move is generated that violates
# one of the boundaries, the robot will not move at all. During the rest of
# the trial, the robot would not move either. With slow speeds, the robot
# will at least move some. [nnw lin5 lin8]
robot_speed            10

# This parameter defines how quickly the robot reaches robot_speed.
# [nnw lin5 lin8]
robot_accel            10

# The percentage of patterns that are known in advance that are trained to
# the matrices. These patterns are of the form
#
#          (th1, th2, 0, 0, 0 [ , 0, 0, 0] ) -> (0, 0, 0)
#
# Actually this says that if the target position has been reached exactly,
# no movement is done. The optional part is required if the input adjustment
# method is disabled. [lin5 lin8]
known_pattern_perc 50

# The neural network approach is computationally very expensive. While the
# training progresses, more and more patterns are collected for each node
# after each trial, that have to be trained. In particular the root-node
# collects as much patterns as there are bins, a number which raises very
# quickly. nnw_train_limit sets a maximum of number of patterns that should
# be learned to a node after each trial. If more patterns are collected for
# a node than nnw_train_limit, around nnw_train_limit patterns (sometimes
```

```
# slightly more, sometimes slightly less, but on average nnw_train_limit)
# will be trained to that node.
nnw_train_limit      300

# Another parameter in order to speed up neural networks. Nodes will only
# be updated after at least nnw_train_delay additional patterns have been
# added since the last updated. This way, it is avoided that every time a
# node receives one or a few new learning patterns, the node is trained
# again.
nnw_train_delay       15

# The two variables below are used when perceptrons (ie. the matrices) are
# turned off and the eye-hand mapping function is approach with feed-forward
# networks. [nnw]
wrange 0.1
hidden_units 5
```

A.3 Data structures

A range of data structures exist to accommodate among others the tree, bins, learning patterns and matrices. The key data structure is the *node* which is, as the name implies a node in the tree (see figure **??**). To every node in the tree, the following data structures are linked:

- The parent node (except for the root node),
- The child nodes,
- A circular linked list of perceptrons,
- A single linked list of bins,
- An error monitor,
- An error monitor for each subspace that has not yet received a child.

Each perceptron consists of a matrix containing the sums of the left-hand side of equation (**??**), as well as the right-hand side of that equation: 'matrix' and 'rhs' respectively. The solution for the current perceptron, i.e., for $\mathcal{N}$, is stored in the node data structure itself: 'solution'. The bin data structure contains a unique identification 'bin_id' which is an array of branch numbers (see equation (**??** and **??**). Furthermore, it contains a flag 'full_flag' and the number of the next slot to be filled when a new learning pattern arrives 'current_pat' to facilitate wrap around when the bin is full and to be able to determine the number of learning patterns a bin contains. Lastly, each bin contains pointers to the patterns itself. These pointers may point to either dynamically allocated memory or to positions in the `pat_file`, depending on which method of storage the user has selected with the `use_pat_file` entry in the configuration file. 'child_error' and 'abs_error' point to error_monitor structures. These data structures very efficiently calculate the average of numbers that are stored in them using appropriate calls to functions that manage error monitor structures. 'average_nr' holds the amount of numbers that should be averaged upon. 'error_cnt' holds the number of the next free slot in the dynamically allocated array 'errors[average_nr]'. 'error_sum' holds the sum of these errors. At the point 'average_nr' numbers have arrived, 'error_sum' will be updated such that it reflects the sum of the 'average_nr' most recently arrived numbers. To be able to do so, the most recently arrived 'average_nr' numbers are stored in 'errors'.

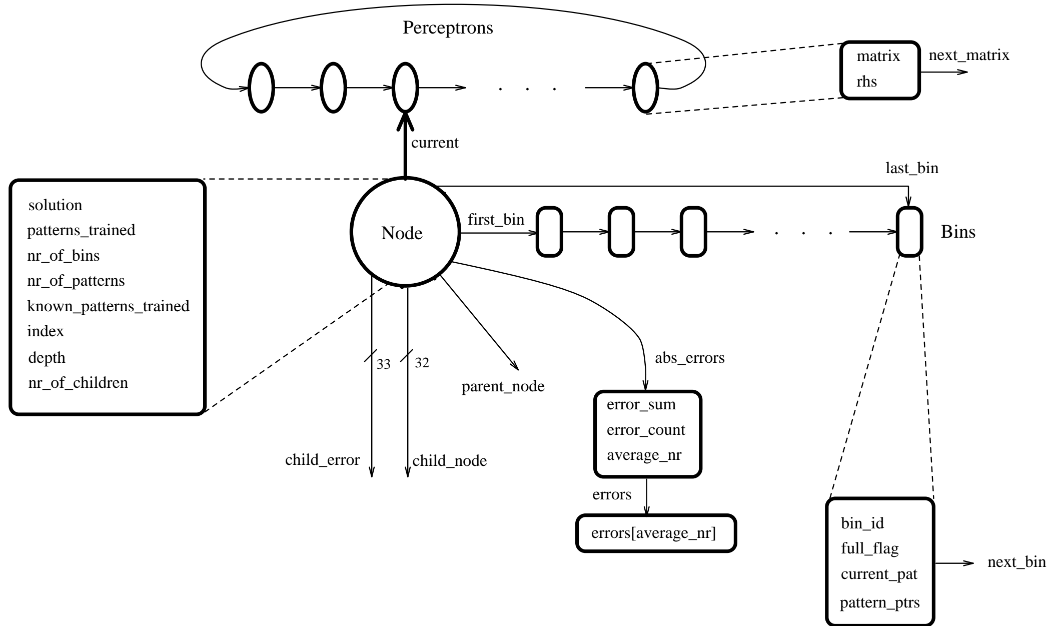


Figure A.1: Internal representation of a tree-node. To each node a circular linked list of perceptrons is linked: 'current' points to the perceptron that is currently used to calculate the solution of $\mathcal{N}$. Also, a singly linked list of bins is connected to the node: 'first_bin' and 'last_bin' point to the first and last bin respectively. For each node a maximum of 32 child nodes can exist: 'child_node[32]'. Each child node has a pointer to its parent node: 'parent_node'. Also, an error monitor is connected, 'abs_errors', that keeps track of the average error. The error is calculated for the most recently `am_error_iters` arrived learning patterns, this parameter has an entry in the configuration file. If a subspace i has not yet received a child, the average error that a node generates for that subspace will be stored in an appropriate error monitor: 'child_error[i]'. A node also does some bookkeeping: 'patterns_trained' revers to the number of patterns that are trained to the perceptrons connected to it. This number is used to manage the 'current' pointer to the perceptrons. 'nr_of_bins' holds the number of bins connected to it, 'nr_of_patterns' holds the total number of learning patterns stored in these bins. 'known_patterns_trained' holds the number of known patterns trained to the perceptrons, 'index' is the branch number, 'depth' holds the depth of the tree in the node, and 'nr_of_children' finally holds the number of children of the node. The other data structures displayed are explained in the text.

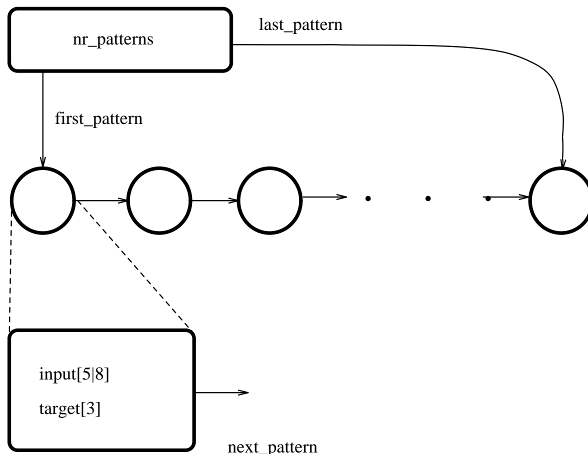


Figure A.2: Internal representation of a pattern-list. 'nr_patterns' holds the number of learning patterns, 'first_pattern' and 'last_pattern' point to the first and last learning pattern respectively, and a node in this list contains an input vector 'input' which is either of size 5 or 8 depending on the dimensionality of $\mathcal{F}$, and a target vector 'target' of size 3, which corresponds to the number of joints.

Finally, an intermediate pattern list structure exists, which is depicted in figure **??**. Functions to support this structure exist. It is used to store lists of learning patterns that are collected for various reasons during the learning process and dynamically allocated.

A.4 Complexity and internal mechanisms

The plots of the time used reveal that the number of trials per time-unit is independent of the size of the tree. The reason for this simply is that there are no recursive routines that walk the tree. The system has been programmed such that tree-walking is unnecessary. Trivial things like the number of nodes in the tree as well as the number of bins and learning patterns are continually updated on a per level basis. As soon as a node receives a learning pattern, several things happen. First of all, the pattern is stored and the system increases the count of the number of learning patterns for the level at which the node exists by one. If a bin is needed to hold that learning pattern, the number of bins is also increased by one for that level, just as well as the 'patterns_trained' and 'nr_of_patterns' parameters within the node data structure. Then, the system checks if the node needs to be split or merged. After all, the new learning pattern may cause the conditions for a split or merge of some subspace to be fulfilled. None of these actions require a tree-walk. Also, the number of perceptrons that have to be trained at some moment in time only depends on the depth in the tree, since each learning pattern will be propagated to all of its ancestors. It is however more or less independent on the number of nodes in the tree. Also, training a perceptron consists of updating it and recalculating the solution for each joint, which are very cheap operations. The time used for training perceptrons takes only about 50% of the total computational resources used by the system. This excludes time needed for bookkeeping the several error-structures, storing patterns and checking for split and merge, to name just a few things. Therefore, the amount of time used per trial is virtually independent of the size of the tree, which is an asset, since the system will not show any slow down even after as much as 50,000 trials. Would on the other hand feed-forward networks be used instead of perceptrons, the

increasing number of networks to be trained caused by the increasing depth of the tree becomes a significant factor, because of the huge amount of computational resources that are needed by feed-forward networks. Running a system for 50,000 trials with feed-forward networks is estimated to take about a year even on a Sun SPARCstation 20.

Bibliography

[1] Arjen Jansen, "Neural approaches in the approximation of the inverse kinematics function: A Comparative Study," University of Amsterdam, Vakgroep Computerarchitectuur, March 1993, *Masters Thesis*

[2] P. van der Smagt, "Simderella: a robot simulator for neuro-controller design," *Neurocomputing*, Vol. 6, No. 2, 1994, *Elsevier Science Publishers.*

[3] B. J. A. Kröse, P. van der Smagt & F. C. A. Groen, "A one-eyed self-learning robot manipulator." In G. Bekey & K. Goldberg (Ed.), *Neural Networks in Robotics*, Kluwer Academic Publishers, (1992), 19–28.

[4] A. Jansen, P. van der Smagt & F. C. A. Groen, "Nested Networks for Robot Control." In A. F. Murray, *Neural Network Applications.* Kluwer Academic Publishers (1994).

[5] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics* 43 (1982), 59–69.

[6] T. Kohonen, "Self-organization and Associative Memory," Springer-Verlag, Berlin, 1984.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, "Numerical Recipes: The Art of Scientific Computing," Cambridge University Press, Cambridge, 1986.

[8] F.C.A. Groen: personal communication