The Development of a Genetic Programming Method for Kinematic Robot Calibration

Jens- Uwe Dolinsky

Liverpool John Moores University

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

March 2001

Abstract

Kinematic robot calibration is the key requirement for the successful application of offline programming to industrial robotics. To compensate for inaccurate robot tool positioning, offline generated poses need to be corrected using a calibrated kinematic model, leading the robot to the desired poses. Conventional robot calibration techniques are heavily reliant upon numerical optimisation methods for model parameter estimation. However, the non-linearities of the kinematic equations, inappropriate model parameterisations with possible parameter discontinuities or redundancies, typically result in badly conditioned parameter identification. Research in kinematic robot calibration has therefore mainly focused on finding robot models and appropriate accommodated numerical methods to increase the accuracy of these models.

This thesis presents an alternative approach to conventional kinematic robot calibration and develops a new inverse static kinematic calibration method based on the recent genetic programming paradigm. In this method the process of robot calibration is fully automated by applying symbolic model regression to model synthesis (structure and parameters) without involving iterative numerical methods for parameter identification, thus avoiding their drawbacks such as local convergence, numerical instability and parameter discontinuities. The approach developed in this work is focused on the evolutionary design and implementation of computer programs that model all error effects in particular non-geometric effects such as gear transmission errors, which considerably affect the overall positional accuracy of a robot. Genetic programming is employed to account for these effects and to induce joint correction models used to compensate for positional errors. The potential of this portable method is demonstrated in calibration experiments carried out on an industrial robot.

Acknowledgements

I am not going to thank anybody - because I did it all myself.

Milligan, Spike (1918)
British comic actor and author.
On receiving the British Comedy Award for
Lifetime Achievement in 1994

I would like to express my gratitude to my director of studies Dr. Gary. J. Colquhoun at Liverpool John Moores University for his guidance, help and support throughout the course of study within the last years. His wisdom, experience and knowledge, especially of administrative mechanisms, burdens and resources within the university proved extremely beneficial for my work.

My thank also goes to my supervisor Dr. Ian D. Jenkinson who, obviously being one of the busiest members of staff at Liverpool John Moores University, has always found some time for formal and informal discussions about my research. Amazingly, it was often the (lingual) misunderstandings we had during those discussions, which sparked new ideas and started me off exploring (and exploiting), quite analogously to the genetic programming paradigm used in this thesis, new areas of research.

Also, I would like to thank the technicians of lab B.4 at JMU who provided tea, interesting conversations and most importantly help when I was fighting with the robot during my numerous measurement and calibration sessions in the laboratory.

I gratefully acknowledge the support given by users of the Internet discussion groups comp.ai.genetic and comp.robotics.research.

Finally, I would like to thank for the support given by the School of Engineering of Liverpool John Moores University.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preamble

In the development of a new technology one often underestimates the complexity of the task and the direction, duration and scope of the research effort involved. An example in computer science context is the attempt to create artificial intelligence, which has been pursued ever since the first electronic computers became available in the mid 1940's. After the initial euphoria in recognising the potential of computing, the development of computational concepts corresponding to human intelligence based on cognition, intuition, experiences, mentality, emotions and feelings proved difficult. The high complexity of processes attributed to human intelligence, the limited understanding of them, the lack or insufficiency of abstract mathematical formalisms to describe them, and limited computational resources permit only highly simplified modelling.

Similarly, whereas the development of computer hard- and software has boomed in the last two decades, the predictions made in the 1970's to have reliably accurate programmable industrial robots by the 1990's however turned out to be too optimistic. Again, a reason for failed expectations is the complexity of the problem, namely the complex mechanical structure of the robot whose physical properties are only insufficiently captured in the controller model to enable the robot to perform offline programmed operations accurately. This compromise of using a simplified model, in fact only kinematic properties are generally covered, is made due to

economic and computational efficiency considerations. On one hand the robot model must be kept simple[1] in order to enable efficient computation by the controller in the presence of real time constraints. On the other hand due to economic constraints in manufacturing robots the model parameters in the controller are initialised per default with nominal parameter values (A one-size-fits-all strategy). However, these nominal parameter values have been determined in the design phase of the robot, and only incompletely reflect the real state of the individual mechanical system. In fact, due to manufacturing tolerances the mechanical structure of each robot has its individual set of parameters (also referred to as its *signature*), which varies between robots of the same model type. Over time wear and tear also cause parameter drift. In addition the still limited, noisy information delivered by sensors on the performance of the robot within its working environment contributes to the complexity of robot control. With the robot controller having imprecise information about the workcell and robot hardware it is operating, the result is inaccurate positioning of the robot tool, which strongly limits the applicability of offline programming. To overcome this problem is the objective of robot calibration, which is essentially concerned with the identification of more accurate robot models enhancing the controller software and thus increasing the positional accuracy of a manipulator.

1.2 Contribution to knowledge

This thesis contributes a novel portable static kinematic calibration technique for industrial robots, which is based on the recently developed genetic programming (GP) paradigm. The contribution and the advantages of this approach compared to conventional traditional calibration methods are summarised as follows:

- The design and synthesis process of the joint correction models is fully automated. The genetic programming algorithm establishes structure and parameter values of joint correction models based on measured data.
- Calibration is carried out by symbolic rather that numeric regression. The final solution is a symbolic mathematical representation of the correction models.

[1] time-invariant parameters only; ignoring dynamical effects and noise

Thus, unlike with other numerical approaches utilising artificial neural networks for example, not only is the required error compensation obtained but also a symbolic description of the complex joint related effects, which enables further mathematical analysis.

- Genetic programming, as it is used in this work, does not require numerical parameter identification e.g. gradient search, which would be bound to fail since the evolved equations are highly non-linear and discontinuous in their parameters.
- Due to the nature of genetic programming as being a stochastic search technique the proposed method has the potential to establish a globally optimal calibration model.
- The new calibration method has been experimentally examined in laboratory trials on a Unimation PUMA robot. The results of these successful experiments show the potential of this new method.

1.3 Contents of this thesis

After a general introduction to robot calibration and its terminology in Chapter 2 conventional kinematic robot modelling and calibration methods are reviewed and their limitations, which motivated this research, are outlined. Concluding the chapter, the scope of this work is defined and the ideas underlying the developed calibration method are presented. Chapter 3 describes evolutionary computation principles implemented by the developed calibration method. In particular genetic programming is introduced as an effective technique for automatically generating computer programs that solve a variety of tasks. Chapter 4 brings genetic programming into robot calibration context as being the fundament of a new static kinematic calibration method, in which it is applied to evolve joint correction models as components of the overall kinematic model of an industrial robot to improve its static positional accuracy. This new evolutionary calibration technique is developed in this chapter and the mathematical underpinnings and algorithms are described in detail. Also the principle design of the calibration system implementing the developed calibration algorithms is outlined. The implementation of this calibration system design in software is described in Chapter 5. The potential of this new

calibration method is demonstrated in Chapter 6, which includes experimental results from the calibration trials on a laboratory PUMA 761 robot. Chapter 6 also contains a critical discussion of the research. The thesis concludes in Chapter 7 with an outlook and suggestions for further work.

Chapter 2

Robot calibration

This chapter reviews robot calibration and its requirement in context of offline programming. Firstly, a general introduction to the calibration terminology is provided followed by a description of the general phases involved in the calibration process. Kinematic modelling as the basis of robot calibration is then reviewed and the limitations of current calibration approaches are outlined. Concluding the chapter the scope of this research is defined.

2.1 Context and terminology

An industrial robot (IR) is defined by the Robotics Industries Association (RIA) to be *"a re-programmable, multifunctional manipulator designed to move material, parts, tools, or specialised devices through various programmed motions for the performance of a variety of tasks*". The mechanical structure of an industrial robot, also referred to as the *manipulator*, is made up of a sequence of rigid links that are interconnected by *prismatic* or *revolute* joints enabling relative motion (driven by actuators) of neighbouring links. This composite sequence forms an *open kinematic chain* for most manipulator types (including the Unimation PUMA 761 used for experiments reported in this work). If the ends of this chain interconnect, the manipulator is said to form a *closed kinematic loop*.

Important quality and performance characteristics of an industrial robot are its *repeatability* and *accuracy*. Repeatability in this context is defined as the *precision*

with which the manipulator is capable to return to a reference point in the workspace[2]. Accuracy is the measure of deviation in terms of position and orientation of the robot tool (also termed *end- effector*) between programmed path and actual achieved path. While nowadays robots have a very high repeatability their positional accuracy is relatively poor. In fact, the accuracy is up to 10 times of the typical repeatability of industrial robots, which is about ±(0.1 – 1.0) mm [12][82]. Both accuracy and repeatability are affected by the *control resolution*, which is the smallest programmable motion of the manipulator.

2.1.1 Robot programming

The methods, an industrial robot can be "instructed" or programmed to perform tasks, can be classified into (i) teaching methods and (ii) high level programming.

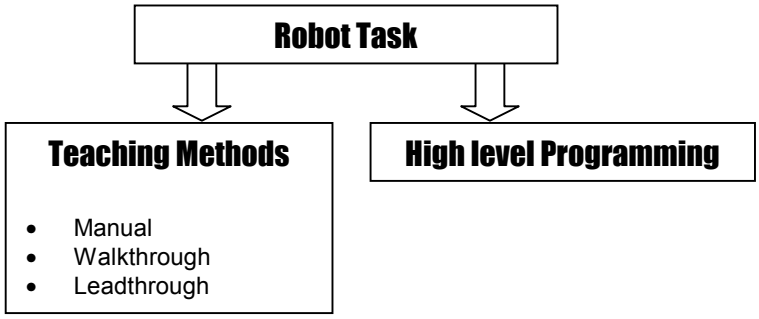


Figure 2-1: Robot programming methods

(i) Teaching methods are procedures that require the manipulator to be moved manually to all desired locations in the workspace by an operator. The simplest method is the manual method, which is more considered a set-up procedure (the adjustment of mechanical stops, cams and switches) and addresses low technology robots (pick and place units). The walkthrough or manual leadthrough method is a continuous-path programming technique in which the robot arm is manually moved along the desired path and the resulting trajectory recorded simultaneously. This approach however requires a significant amount of data memory and often utilises

[2] The term workspace or *working envelope* refers to the area that is accessible by the manipulator.

disk storage. The leadthrough method, also sometimes called powered leadthrough, is the most common teaching technique in which the operator takes the robot arm to the target locations typically using a handheld teach-pendant. The locations are stored in the controller memory as they are taught. After completing the teaching procedure, the manipulator can playback the paths along those recorded locations very precisely benefiting from its high repeatability.

(ii) A more advanced and portable method is to program the tasks to be performed in a high level programming language such as VAL II [72]. The support of general concepts of structured programming[3], built-in geometric entities such as frames and locations points combined with specific robot motion commands makes these programming languages efficient tools for developing complex robot tasks e.g. by aggregating primitive robot instructions to high-level commands. However, the support of those concepts requires a higher degree of sophistication from the robot controller then teaching methods. Firstly, the controller needs to deal with the overhead of interpreting and executing programs. Secondly, robot end-effector positions in the program may be specified in geometric terms relatively to the base frame of the robot for example. This requires the controller to implement an inverse kinematic model of the manipulator to convert these 3D poses into joint configurations, which in turn implies accurate knowledge about the parameters of the mechanical structure of the robot.

The fundamental difference between teaching methods and high level programming is that with teaching methods the controller is provided with the physical goal and the information how to reach this goal. This information is stored (or recorded) in terms of current status of the control system (i.e. the joint configuration) of this particular robot. During the execution of the program these previous states can be restored very precisely leading to a high repeatability. In high level programming the controller is provided with a "soft goal" (rather than a physical goal) specified by the programmer as a position in e.g. Cartesian co-ordinates. Since no further information is provided as to how to reach this position, an inverse kinematic model of the manipulator is needed to convert this position into

[3] Sequencing, selection and iteration.

a corresponding joint configuration. Inaccuracies in this model result in poor absolute positional accuracy of the manipulator.

2.1.2 Offline Programming Systems

Flexible and effective robot programming has become an important issue in industry. In the past industrial robots were mainly programmed manually by being taught individual tasks on-line as described in section 2.1.1. However, the increasing complexity of tasks such as riveting or spot welding in car or aerospace industries makes such a teaching procedure extremely time and therefore cost intensive. The conceptually better approach is to minimise human intervention in operating robots by using Offline-Programming Systems (OLP), which enable the design, generation and validation of robot programs without utilising the physical robot. OLP systems use models of the robot and workcell to create a virtual shop-floor environment in which robot tasks can be simulated. The advantages and benefits of OLP systems are summarised as follows:

- Easy design of complex tasks (using imported CAD data)
- Reduction of programming time and production plant downtimes (robot hardware is not involved in the programming process, thus production and programming new tasks can run simultaneously)
- Increased production flexibility
- Failsafe debugging, optimisation and validation of tasks (collision and reachability checks in simulation, no risk of hardware damage)
- Simplification of process optimisation
- Fast validation of tasks (simulation can be run considerably faster than the robot hardware)
- No risk to human health (programming in a comfortable environment, presence of programmer in physical workcell is not required)

Quite a few commercial OLP systems are now available with an increasing number of PC based solutions such as Workspace® [82], DELMIA/IGRIP® [19]

(used by e.g. Boeing) and more recent products such as FAMOS®[30], RobotStudio® [1] by ABB and Ropsim® [15] by CAMELOT.

The following components are typical for all OLP systems:

- **Integrated CAD system**: constructive solid geometry, library of standard 3D primitives, CAD data import facilities (e.g. DXF, IGES)
- **Modelling and simulation module**: interactive kinematic modeller, modelling of robot dynamics, discrete event simulation
- **Visualisation module**: graphical representation of results, solid 3D rendering, animations of robot and/or production plant in real-time
- **Library of standard robots models**: models describe individual kinematic and dynamic characteristics and contain path planning algorithms
- **Robot calibration module**: set of methods for numerical optimisation
- **Interpreter and code generator for advanced robot programming languages**

Figure 2-2: The OLP system IGRIP®

Limitations in the application of OLP systems to robot programming are still imposed by the poor positional accuracy of industrial robots and by the lack of exact modelling of those inaccuracies. The deviations between idealised simulation in a

virtual environment and the real system cause the OLP system to generate robot poses with large positional errors. Also path-planning algorithms used within OLP systems are often different to or simplified versions of those used by the robot manufacturer, which results in unreliable information about cycle time and possible collisions.

2.1.3 The role of robot calibration

In order to avoid positional errors of the robot tool positional calibration needs to be applied, which can in general be categorised into hardware and software oriented methods (Figure 2-3). More accurate positioning of the robot tool can be obtained by appropriately modifying the mechanical structure of the robot for example by replacing worn-out components by new more accurately manufactured components.

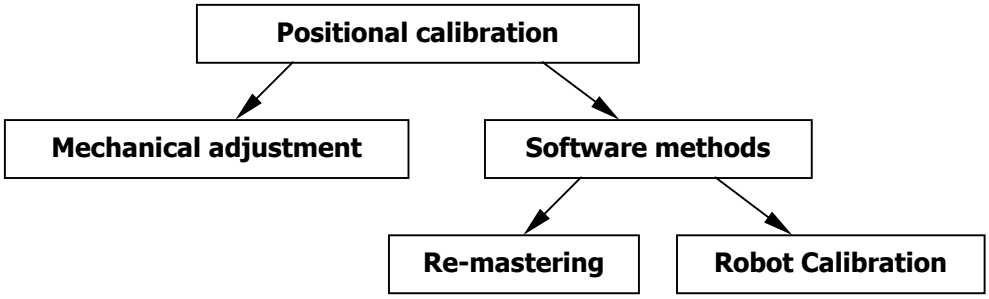


Figure 2-3: Positional calibration principles

The software-oriented methods address positional accuracy on robot controller or robot application level. Principally, it is possible to adjust robot accuracy by modifying appropriate parameters of the controller software. However, since robot manufacturers usually do not document the algorithms and data structures used by the controller (the 'black box' policy), this method is, apart from being not portable, fairly limited. However, a generally supported software method falling into this category is known as *re-mastering*. Re-mastering is a method where a joint is moved to a defined position (usually the zero position designated e.g. by a mark on the neighboured link). The controller software is then updated with this new reference position by issuing the appropriate command to the controller.

Robot calibration as the second software-oriented method to adjust robot accuracy applies position compensation on robot application level based on a calibrated model (Figure 2-4). Robot poses (only the 3 positional components) defined in programs, which may have been generated by OLP systems are modified by subtracting the expected positional error estimated by an accurate calibrated model (see section 4.2 and Figure 4-2). In this way false target poses are produced with the objective to compensate for the positional error. The deviation of the manipulator at these altered poses eventually leads the robot to the desired positions. This process is also known as task space compensation since the positions and the applied corrections are defined in task space (as opposed to joint space).

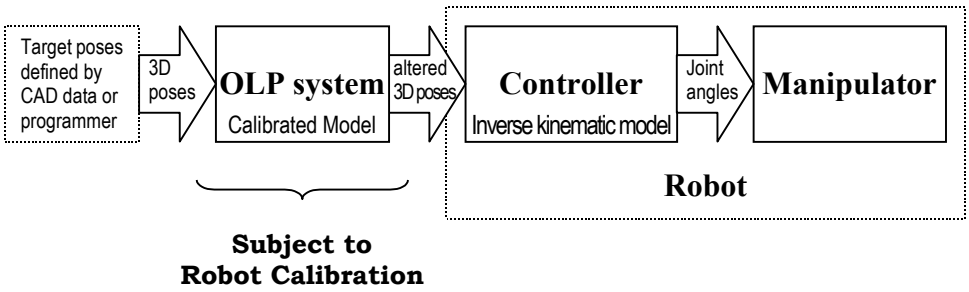


Figure 2-4: The scope of robot calibration in offline programming

Robot calibration is a term associated with a set of software methods aiming at the identification of accurate robot models used with the objective to increase the positional accuracy of the robot [67]. There is a distinction between *static* and *dynamic* calibration methods [12]. Static calibration aims at the identification of accurate models covering all physical properties and effects that influence the static (time invariant) positional accuracy of the manipulator. Dynamic calibration builds upon the results of static calibration and addresses the identification of models describing motion characteristics of the manipulator (forces, actuator torques) and dynamic effects that occur on a manipulator such as friction and link stiffness etc. In order to enable dynamic calibration, measurements of motion and forces of the manipulator are required. However, the difficulties in accurate tracking of these properties throughout the robot workspace and the complex problem of simultaneous

identification of e.g. mass[4] and friction parameters still limit the applicability of dynamic calibration.

This work is concerned with static kinematic robot calibration, which is typically carried out using the 4 following steps:

1. Derivation of a suitable kinematic model usually based on prior engineering knowledge (providing a model structure and nominal parameter values).
2. Measuring end-effector location of the manipulator in several positions.
3. Identification of the model parameters (numerical fitting usually based on least squares methods) based on the measurements.
4. Implementation of the identified model.

2.2 Kinematic modelling

Static robot calibration typically uses parametric models of the manipulator kinematics to find the true relationship between joint configurations and poses of the robot end-effector. Most work in this area has reported on forward calibration methods [23][37][43][47] where calibration is applied to the forward kinematic model:

$$\mathbf{y} = \mathbf{f}(\theta, \phi)$$

which computes the end-effector pose $\mathbf{y}$ (position and orientation) from the joint configuration θ using the equations in $\mathbf{f}$ depending on parameter vector ϕ to be calibrated. The inverse kinematic calibration procedure attempts to identify the parameter vector ϕ using the inverted kinematic model $\theta = \mathbf{f}^{-1}(\mathbf{y}, \phi)$. Inverse calibration is in general more difficult because it typically requires the model $\mathbf{f}$ to be analytically invertible, which may not be the case with very complex models and in particular with models of highly redundant[5] manipulators.

Important issues for the development of accurate parametric kinematic robot models for robot calibration are *proportionality* and *completeness* [27]. A kinematic

[4] Mass parameters of a link can however be statically identified using joint torque sensors [54].
[5] Redundant manipulators can reach a certain pose using different joint configurations.

model is defined as proportional[6] if small changes of the physical properties can be represented by small changes of related model parameters. A kinematic model is said to be complete if all kinematic properties of the manipulator are represented by corresponding independent model parameters. In this case all possible kinematic configurations of the manipulator can be sufficiently described. With incomplete kinematic models the number of model parameters is usually smaller than the number of kinematic properties of the manipulator. The contained parameters then account for modelled properties as well as unmodelled effects. Hence there is no proper relationship between physical and model parameters. An identification algorithm might be able to find optimal parameter values. However, these values are optimised for this particular incomplete model and may not reflect the physical properties of the robot. In order for a kinematic model to be complete it is required to sufficiently describe geometric properties as well as non-geometric effects of the manipulator (see Table 2-1).

Geometric Parameters	**Non-geometric effects**
• link length • link twist- joint axis orientation • joint encoder offsets	*Joint related*: • Gear transmission errors (e.g. tooth errors) • Joint compliance • Joint eccentricities, bearing wobble • Gear Backlash • Joint cross coupling *Link related:* • static deflection • thermal expansion *Encoder related:* • Non-linear transfer function • Coupling • Hysteresis

Table 2-1: Physical properties to be considered by an accurate kinematic model

[6] Proportionality is sometimes termed "model continuity" due to its similarity to the mathematical concept of continuity [12][37].

Geometric parameters are usually assumed to be time-invariant, which is convenient for setting up a compensation strategy to improve positional accuracy. Errors in geometric parameters have been reported to have the largest contribution to positional error. For example in experiments with Automatix AID-900 Robots almost 90% of the RMS (Root Mean Square) error was caused by joint angle offsets [47]. On a TH8-ACMA six-axis robot the calibration of geometric parameters resulted in accuracy improvements from 3 mm to 0.69 mm. Model refinements accounting for non-geometric effects achieved further accuracy improvements to 0.58 mm [14]. Non-geometric effects [76] such as gear backlash and tumbling are difficult to model since they vary with manipulator pose and payload. However, their contribution to the positional error of the manipulator cannot be neglected if high positional accuracy of the manipulator is required.

2.2.1 Geometric modelling

The most popular method of modelling robot kinematics is serially composing link models that are based on the Denavit-Hartenberg (DH) parameterisation [34]. These link models use only four geometric parameters per link to describe the relative displacement between co-ordinate frames of neighboured links. Hence kinematic models based on DH parameters are very compact and have therefore been commonly implemented in controller software[7]. However, for calibration purposes pure DH models do not fulfil the requirements of completeness and proportionality [84]. Four parameters are not sufficient to describe any arbitrary displacement of two consecutive link frames. Since the DH parameterisation relies on the existence on a common normal of neighboured joint axes, it is not well defined in configurations where neighboured joint axes are at or near to parallel[8]. Hence the identification is ill-conditioned and will either fail or result in meaningless parameter values. Addressing this issue Hayati [41] proposed a modification to the DH model by

[7] Denavit-Hartenberg parameters (see also Appendix section A.2) are commonly used by robot manufactures to document the geometric properties of their robots (see for example the equipment manual of the PUMA 761 [75]).

[8] Near to parallel neighboured joint axes certain model parameters are very sensitive to small physical changes (non-proportionality), whereas the model continuity constraint is violated with the transition from the near-to-parallel to parallel case.

introducing an alternative parameterisation, which is however not well defined for nearly perpendicular joint axes. Hollerbach [43] has therefore suggested applying a complementary mix of DH and Hayati parameterisation by using Hayati parameters whenever the corresponding DH parameters are not well defined and vice versa. Other researchers have proposed geometric extensions to the conventional DH model. Stone [73] developed the S-model by adding 2 parameters to the DH model, which results in a complete, but not proportional parameterisation [84]. As with Hayati parameters the S-model parameters can be converted back into DH parameters.

An incomplete model can be made complete by appropriately adding a certain number of parameters. This however impairs the computational performance of the model and may affect the identifiability of the model parameters. A complete and proportional (because singularity free) parameterisation is the CPC (Complete and Parametrically Continuous) model [84], which is based on the DH model extended by 2 parameters. Other complete models are the zero-reference model [59] and the Sheth- Uicker model [29][37] in which redundant parameters have to be removed prior to calibration (or held constant during calibration), which however is not always considered to be a straightforward task [76].

2.2.2 Examples of non-geometric models

Non-geometric effects are usually modelled by adding terms or more complex components to the overall geometric model of the manipulator. Since non-geometric effects are primarily due to joint related characteristics [76] the most common model adopted (see [37][43]) is a simple linear joint correction model:

$$\Theta = \mathbf{k}\theta + \gamma$$

where the effective joint angle Θ is computed from the joint angle sensor reading θ depending on the joint angle zero offset γ and joint transmission gain $\mathbf{k}$. This model is applied to each joint adding 2 more parameters per joint to be calibrated to the overall model. In other research [81] different joint models are applied to selected joints (1-3):

$$\mathbf{\Theta_1} = \theta_1 + \mathbf{P_1}\cos(\theta_1)$$
$$\mathbf{\Theta_2} = \theta_2 + \mathbf{P_2}\cos(\theta_2) + \mathbf{P_3}\sin(\theta_2) + \mathbf{P_4}\cos(\theta_2 + \theta_3^*)$$
$$\mathbf{\Theta_3} = \theta_3^* + \mathbf{P_5}\sin(\theta_3^*) + \mathbf{P_6}\cos(\theta_3^*) + \mathbf{P_7}\cos(\theta_3^* + \theta_2) + \mathbf{P_8}\sin(\mathbf{load})$$

with $\mathbf{\Theta_i}$ being the effective i[th] joint angle, θ_i the i[th] joint angle sensor reading, $\mathbf{P_i}$ the i[th] parameter to be determined and $\theta_3^* = \theta_3 - \pi/2$.

A different non-parametric approach has been proposed by Everett [29] where each geometric model parameter is enhanced with a Fourier Series (FS) in order to model (or approximate) the influence of particularly periodically occurring non-geometric effects upon these model parameters. Adopted by Vincze [77] a Fourier series was used to describe effects such as tumbling upon all geometric parameters:

$$\mathbf{a}(\mathbf{q}) = \mathbf{a}_{no} + \mathbf{\Delta a} + \sum_{j=1}^{n_{sc}} (\mathbf{a_{js}}\sin(j \cdot \mathbf{q}) + \mathbf{a_{jc}}\cos(j \cdot \mathbf{q}))$$

with $\mathbf{a}_{no}$ being the nominal parameter value, $\mathbf{\Delta a}$ the geometric error, $\mathbf{q}$ the joint variable and $\mathbf{a_{js}}, \mathbf{a_{jc}}$ the Fourier coefficients to be determined. It was found in these experiments that second and third order FS were capable of reducing tumbling errors down to axis repeatability level, which is the theoretical calibration limit [12].

Other non-geometric models include backlash (for example $\theta_b = p_b\mathrm{sign}(M)$ [76] with θ_b being the joint backlash, p_b a backlash parameter and M the momentum at the joint) or errors of transmission ratio (e.g. polynomials or Fourier series). A comprehensive list of non-geometric errors and their models can be found in Vincze et.al.

Vincze [79] also introduced a deterministic method of automatically generating robot models (based on his SYNE-axis description: SYstematic Non-redundant and Extendible). This method composes the calibration model from a given geometric description of the manipulator and given non-geometric models to be used. To achieve non-redundancy of the composed model and to improve its accuracy, deterministic rules are applied to eliminate redundant parameters. If the accuracy after parameter identification is not sufficient the modelling procedure may be repeated using a different set of non-geometric models.

2.2.3 Alternative modelling techniques

Due to the complexity of the positional error of industrial robots it is common practice to approximate the error rather than modelling it explicitly by developing parametric models. Functional approximation theory is a well established mathematical discipline, which provides a variety of approximation models and methods based on uni- and multivariate polynomials, splines, Bezier curves, wavelets, Fourier series, artificial neural networks[9] etc. An approximation model is chosen according to the characteristics of the data to be approximated in a certain interval, and finally fitted (or trained) to this data. However, approximation models are inherently non-parametric, i.e. there is no semantic relationship between model parameters[10] and physical properties of the data to be approximated. Hence these models are only valid within the interval they have been trained in and are weak or not capable of extrapolating or generalising beyond these interval boundaries. However, for most practical applications they are generally assumed to be adequate. Applications of approximation models utilising Fourier series, polynomial functions and artificial neural networks have been outlined in sections 2.2.2 and 2.2.4.

2.2.4 Kinematic modelling in OLP context

When designing tasks for a particular robot using OLP systems, information from a calibrated kinematic model of the manipulator is used to correct designed (or nominal) poses with the expected positional error so as to compensate for the effective positional error. This correcting pose-to-pose mapping (see also Figure 2-4) can be provided by a sequence of calibrated inverse and nominal forward kinematic model as illustrated in Figure 2-5.

[9] However, Jordan and Rummelhart pointed out [46] that a Backpropagation algorithm cannot learn the inverse kinematics of redundant manipulators because it does not represent a functional relationship (one to many mapping). While forward modelling of those manipulators is straightforward by learning the relation (input: joint configuration vector; output: Cartesian vector) the inverse relation cannot be learned just by swapping output/input vector.
[10] Sometimes referred to as coefficients or weights.

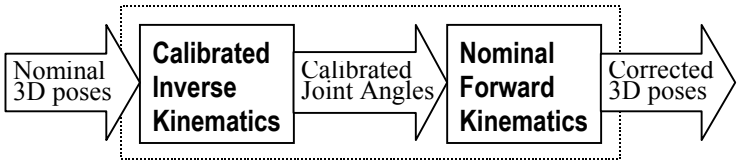


Figure 2-5: Task space compensation in OLP based on a calibrated inverse kinematic model

Designed poses in task space are first transformed by the calibrated inverse model into calibrated joint configurations, which are then transformed back into task space. Alternatively, the calibrated joint angle poses could be fed directly into the robot controller saving two transformations[11]. The inverse calibrated model can be obtained either by calibrating the nominal inverse model or by inverting the calibrated forward model. Both methods require the forward kinematic model to be invertible, which is usually possible for the plain geometric DH model of most manipulators. If so the second method should be preferred since the model inversion introduces equations with even a higher degree of non-linearity (discontinuities) than the forward equations, which would impair the subsequent parameter identification. On the other hand a forward kinematic model that contains complex non-geometric components can usually not be inverted analytically.

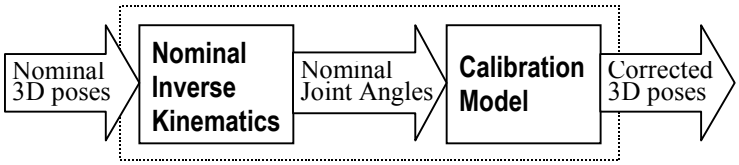


Figure 2-6: Task space compensation in OLP using a mapping that includes a nominal inverse kinematic model

To circumvent the problem of finding an accurate inverse model for the task space compensation model shown in Figure 2-5 a common approach is to use the nominal inverse to convert nominal poses into nominal joint configurations. The relationship

[11] Nominal forward transformation (Figure 2-5) of calibrated joint configurations by the calibration software, and nominal inverse transformation within the controller (see Figure 2-4)

between nominal joint configurations and corrected end-effector poses (i.e. the forward model) is then made subject to calibration as illustrated by Figure 2-6.

An inverse calibration solution fitting in this category was presented by Shamma [70]. This method improves the accuracy of the inverse mapping by modelling the error of the nominal inverse model rather than calibrating the inverse model. The error between nominal and calibrated joint configuration for different joints is approximated by polynomial functions, which however have no physical significance. These predefined functions together with the nominal forward model constitute the calibration model shown in Figure 2-6:

$$\mathbf{f_{Cal}}(\theta) := \mathbf{f}(\mathbf{h}(\theta, \xi), \phi_{\mathbf{N}})$$

with $\mathbf{h}$ being the vector of polynomial functions depending on parameters ξ to be calibrated, $\phi_{\mathbf{N}}$ being the vector of nominal kinematic parameters and θ being the vector of nominal joint angles. A similar technique for inverse calibration is described by Zhong [85], in whose work a feed-forward artificial neural network (ANN) is used to compute the corrections for a given joint configuration. The calibration model in Figure 2-6 can for this method be formulated as

$$\mathbf{f_{Cal}}(\theta) := \mathbf{f}(\theta + \mathbf{F}_{\mathrm{ANN}}(\theta), \phi_{\mathbf{N}})$$

with $\mathbf{F}_{\mathrm{ANN}}$ being the artificial neural network to be calibrated (or trained).

A solution based on forward calibration [45] uses ANN's based on Radial Basis Functions (RBF) to map nominal joint configurations to corrections of the end-effector pose. The calibration model in Figure 2-6 is then described by:

$$\mathbf{f_{Cal}}(\theta) := \mathbf{f}(\theta, \phi_{\mathbf{N}}) + \mathbf{F_{RBF}}(\theta)$$

which uses the nominal kinematic forward model and adds the corrections delivered by the RBF networks to establish a corrected pose for a given nominal joint configuration. In this model the RBF networks are subject to calibration.

2.3 Measurements

The second step in robot calibration typically involves the collection of data. To perform open-loop calibration (for closed-loop calibration, see e.g. [43]) on a

kinematic manipulator model a sufficiently large set of data pairs (consisting of end-effector pose and corresponding joint configuration) needs to be sampled using an external measurement device. Different measurement systems are available varying in measurement method (contact and non-contact), the number of captured DOF's[12], accuracy and costs [43]. Typical measurement devices for robot calibration are wire potentiometers, telescopic ball system measured by a radial distance linear transducer (LVDT) [37], interferometer, ultrasonic systems [11], proximity sensors, imaging laser tracking systems [78], single and stereo camera systems, magnetic trackers, (stereo) theodolites and cable driven systems [82] etc.

In order to ensure a good performance of the parameter identification procedure a sufficiently large set of data samples needs to be recorded where the poses have to be selected throughout the workspace in a way that guarantees the best observability[13] of all parameters to be calibrated [43]. To identify the influence of a parameter on all DOF's it would in general be beneficial to involve full pose measurements. In practice, however, appropriate measurement devices are fairly expensive, relatively slow, and difficult to set up. Alternatively, calibration can be performed using only position measurements, since all kinematic parameters of a manipulator may be identified based on position measurements if the measured points are not located along the tool axis [25].

In the experiments reported in this thesis the Robotrak measurement device [82] (position measurements only, see also Appendix section A.1) was used because it is robust, easy to set up and to use, particularly when a large set of measurement data samples needs to be recorded.

2.4 Parameter identification

Having established the structure of the kinematic model and a set of measurement data, the third step in robot calibration is the numeric identification of the model parameters. Generally used in practice are indirect methods based on gradient search,

[12] From single to 6 Degrees Of Freedom (6 DOF = full pose consisting of 3 position and 3 orientation components).
[13] Parameters must be sufficiently excited by the sampled poses to guarantee a successful identification.

which will be outlined in this section. The notation for the forward kinematic model $\mathbf{f}$ has been adopted from section 2.2:

$$\mathbf{y} = \mathbf{f}(\theta, \phi)$$

where $\mathbf{y} = [\mathbf{p}, \varphi]^{\mathbf{T}}$ is the end-effector pose (consisting of position vector $\mathbf{p}$ and the vector φ of orientation values[14]) computed from the joint configuration θ and $\phi = [\phi_1 \ldots \phi_j]^T$ being the vector of all j model parameters (geometric (e.g. DH) and non-geometric). Identification of ϕ can then be performed by minimising the performance index: $\sum_{i=1}^{n} \mathbf{e}_i{}^T \mathbf{e}_i$ with $\mathbf{e}_i = \mathbf{y_i} - \mathbf{f}(\theta_i, \phi)$

subject to ϕ where $(\mathbf{y_i}, \theta_i)$ is the i^{th} sample of $\mathbf{n}$ measurements. Due to the non-linearity of $\mathbf{f}$ in the orientation parameters and possibly in non-geometric parameters the optimal values in ϕ have to be found iteratively by applying a method such as non-linear least squares optimisation (see e.g. [2]). Mostly indirect gradient-based methods are applied (see [43]) for which the model equations have to be locally linearised using a first order Taylor expansion around the current parameter estimate ϕ_k:

$$\mathbf{f}(\theta_i, \phi_k + \boldsymbol{\Delta}\phi) \approx \mathbf{f}(\theta_i, \phi_k) + \mathbf{C}(\theta_i, \phi_k)\boldsymbol{\Delta}\phi$$

where $\mathbf{C}(\theta_i, \phi_k) = \left.\frac{\partial \mathbf{f}}{\partial \phi}\right|_{\theta=\theta_i;\phi=\phi_k}$ is the parameter Jacobian (or gradient) of $\mathbf{f}$ evaluated at the current joint configuration θ_i using the parameter estimate ϕ_k. Substituting this approximation into the performance index equation yields:

$$\sum_{i=1}^{n} \mathbf{e}_i{}^T \mathbf{e}_i \quad \text{with} \quad \mathbf{e_i} = \mathbf{y_i} - \mathbf{f}(\theta_i, \phi_k) - \mathbf{C}(\theta_i, \phi_k)\boldsymbol{\Delta}\phi = \boldsymbol{\Delta}\mathbf{y_i} - \mathbf{C_i}\boldsymbol{\Delta}\phi$$

which has now turned into a linear least squares problem:

$$(\boldsymbol{\Delta}\mathbf{y} - \mathbf{C}\boldsymbol{\Delta}\phi)^T(\boldsymbol{\Delta}\mathbf{y} - \mathbf{C}\boldsymbol{\Delta}\phi) \text{ with } \mathbf{C} = \begin{bmatrix} \mathbf{C_1} \\ \vdots \\ \mathbf{C_n} \end{bmatrix} \text{ and } \boldsymbol{\Delta}\mathbf{y} = \begin{bmatrix} \boldsymbol{\Delta}\mathbf{y_1} \\ \vdots \\ \boldsymbol{\Delta}\mathbf{y_n} \end{bmatrix}$$

subject to the parameter update $\boldsymbol{\Delta}\phi$. The solution is the Gauss- Newton update $\boldsymbol{\Delta}\phi = (\mathbf{C}^T\mathbf{C})^{-1}\mathbf{C}^T\boldsymbol{\Delta}\mathbf{y}$, which is computed in each iteration to refine the parameter values by applying $\phi_{k+1} = \phi_k + \boldsymbol{\Delta}\phi_k$ beginning with an initial estimate for ϕ. The iteration is stopped if the updates become very small. However, the Gauss- Newton

[14] For full pose measurements **f** contains 6 calibration equations (3 position (x, y, z) and 3 orientation components e.g. Euler angles [34][44]). In case only positional measurements were taken **f** contains the 3 positional equations, which however also depend on the orientation parameters.

solution requires the matrix $\mathbf{C}^{\mathrm{T}}\mathbf{C}$ to be inverted, which might not always be possible. Instead typically the Levenberg-Marquardt update $\boldsymbol{\Delta}\phi = (\lambda\mathbf{I} + \mathbf{C}^{\mathrm{T}}\mathbf{C})^{-1}\mathbf{C}^{\mathrm{T}}\boldsymbol{\Delta}\mathbf{y}$ is applied with $\mathbf{I}$ being the identity matrix and λ being a scalar control parameter. λ is chosen and modified during the iterations in a compromise between invertibility of the matrix $\lambda\mathbf{I} + (\mathbf{C}^{\mathrm{T}}\mathbf{C})$ and convergence speed. While a sufficiently large value for λ enables the matrix inversion it also gives the algorithm the slow convergence characteristic of the steepest descent algorithm. A low value on the other hand will increase the efficiency of the parameter updates towards Gauss- Newton updates. Initially, the value of λ is usually chosen to be large enough to enable the matrix inversion and then steadily reduced while monitoring the invertibility of the matrix.

2.4.1 Problems of numerical identification in kinematic robot calibration

The application of indirect methods such as non-linear least squares optimisation to kinematic parameter identification in general raises a number of issues, which have to be considered to obtain high model accuracy. The non-linearity of the kinematic model equations enforces an iterative parameter search based on local model linearisation. Since the equations of ordinary kinematic robot models are only "mildly non-linear" [68] gradient search can usually be effectively applied. Gradient search algorithms however require the model to be continuous in its parameters. As discussed in section 2.2.1 this is not always the case for the DH and Hayati model and therefore requires the application of special methods such as Levenberg-Marquardt or Singular Value Decomposition (SVD) (see [2]).

The utilisation of complex non-geometric models in kinematic models raises particular problems for the simultaneous parameter identification based on gradient search. For example the continuity of an overall kinematic model is impaired by the introduction of discontinuous non-geometric models. Since the gradient cannot be computed at parameter discontinuities, the search aborts.

For global convergence of the gradient search it is important to provide initial estimates of the parameter values close to the optimum. For complex non-geometric models these estimates however may be difficult to obtain. Global convergence is

also at risk if the gradient is inaccurately approximated by, for example, finite differences, which is a common technique in practice particularly if the model is complex. To ensure proper convergence of the non-linear least-squares optimisation it is also important to apply scaling to task variables and parameters [43]. Since position and orientation errors are combined in ordinary least squares, task variable scaling may be necessary to weight these errors differently to account for different accuracy of measurements. Scaling to parameters has to be applied when they cannot be directly combined in least-squares search [22]. Scaling may also improve the conditioning of the identification.

Another serious problem for gradient search is parameter redundancy. In such a case there are more parameters in the model than necessary for model completeness, hence the model is not minimal (redundant parameters do not increase the accuracy of a model [28]). Parameter redundancy results in linear dependence of columns of the Jacobian and therefore in ill-defined identification. Detecting and removing redundant parameter is important, but not trivial particularly in the case of strong parameter interaction [37].

2.5 Implementation

The last step in robot calibration typically involves all procedures and mechanisms necessary to transfer the calibration results into practice. In offline programming this includes the implementation of a postprocessor that uses information from the calibrated model to perform corrections on positional data in program files which have been generated by OLP systems. Fortunately, basic numerical calibration methods and postprocessors have become built-in components in most of the recent OLP systems. Thus the user is not required to perform this task explicitly.

2.6 Scope of this work

Robot calibration methods have so far been considered a set of parameter estimation methods relying heavily on numerical analysis. Conventionally, robot models are developed by humans based on prior engineering knowledge and

according to certain accuracy requirements and specific constraints (e.g. proportionality), which are related to stability issues of subsequent numerical parameter identification.

This research describes an approach to move away from conventional robot calibration methods based on numerical non-linear parameter estimation methods with their drawbacks such as possible ill-conditioning and local convergence. It introduces symbolic model regression techniques to robot calibration and contributes a novel inverse static kinematic calibration method that merges established kinematic modelling techniques with the recent genetic programming paradigm. It is the aim of this research to show the potential genetic programming has to solve the kinematic calibration problem.

Genetic programming is a problem domain independent stochastic method of automatic programming, which has performed extremely successfully in numerous applications in various areas of computer science, physics and engineering [49]. The application to robot calibration reported in this thesis however is new.

In this work genetic programming is employed to generate joint correction models as parts of an inverse calibration model. Contrary to conventional calibration methods this process of designing and validating a correction model is fully automated and does not require human knowledge as to how to build these models. Only information about primitive model components needs to be provided.

The model generation is performed by symbolic regression. Since there is no iterative numerical parameter identification involved, corresponding stability and conditioning issues are of no concern.

Chapter 3

The principles of evolutionary computation

This chapter reviews evolutionary computation and in particular genetic programming emphasising its application to symbolic regression, which is the basis of the kinematic calibration method developed in this research. Firstly, the chapter introduces the general terminology and principles of computational genetic search and outlines the scope of applications. Finally, the concept of genetic programming is presented as a versatile variant of classical evolutionary algorithms, capable of solving a variety of tasks.

3.1 Introduction

Evolutionary Computation (EC) is a broad term describing a set of problem domain independent computational algorithms. These algorithms attempt to find solutions to problems by implementing a search process, which uses artificial mechanisms analogous to natural evolution. Due to this similarity a whole range of the terminology from biology and evolutionary theory has been adopted to describe the principles of evolutionary computation.

The heart of the EC concept is the *Evolutionary Algorithm* (EA). Rather than being confined to improving a single solution an EA takes advantage of operating on a set or *population* of different candidate solutions (also termed *individuals*). An

individual consists of a set of primitive components (the *genes*), which constitutes the *genome* (or *chromosome* as the carrier of genes) as a kind of construction plan used to build the solution. The data structure of an individual solution that undergoes modification by the EA is referred to as the *genotype*. The representation of a solution during evaluation in the particular problem domain is known as the *phenotype*.

During the evaluation of a population each individual (solution) is assigned a *fitness* value, representing a direct measure of its performance. By applying evolutionary operators such as *selection*, *recombination, mutation* and *reproduction* to the individuals based on their fitness, the EA attempts to produce populations of better performing (or fitter) solutions. The process of producing new individuals (also called *children* or *offspring*) is termed *breeding*, which involves selected *parent* individuals from the population. The EA is said to have completed a *generation*, when the old population has been replaced by a population of offspring (traditional EA), or a number of children equal to the population size has replaced individuals in the population (the steady state EA concept – see section 3.2). Over several generations the evolutionary algorithm stochastically infers improved individual solutions *converging* towards the optimal solution. A population is said to have converged if all its individuals have converged. The convergence characteristic (convergence speed) of a population largely depends on the evolutionary operators applied to the individuals. The solution found by an EA is termed an *evolved* solution.

Two key parameters of a general EA are the population size and the number of generations. In addition a termination parameter may be specified, for example the acceptable fitness of the best individual.

The diversity of individual solutions within a population provides a rich pool of different genotypic material, which is used by the EA to create potentially better solutions (the *exploitation* concept). In fact, this diversity guarantees a wide coverage of the search space. Many different solutions are able to represent many different regions in the search space to be evaluated by the EA. This property is known as *exploration* and makes evolutionary algorithms robust tools for searching for globally best performing individuals particularly in multi-modal search spaces.

3.1.1 Historical development

The basic principles outlined in section 3.1 are common to all implementations of the evolutionary algorithm concept such as: evolutionary strategies (ES), evolutionary programming (EP) and genetic algorithms (GA) as the traditional and genetic programming (GP) as a more recent example [71]. Developed for different purposes these paradigms vary in the way they represent individual genotypes, in terms of the fitness measure applied and in design and implementation of the evolutionary operators. Evolutionary strategies were introduced by Rechenberg [65] in the 1960's as a method of continuous parameter optimisation in hydrodynamic context (i.e. real valued representation of genotypes). Initially, solely selection and mutation were used as evolutionary operators on a single solution only. Schwefel [69] enhanced this approach by the populational concept and the recombination operator. Independently of the work on evolutionary strategies Fogel [31] developed evolutionary programming originally in an attempt to create artificial intelligence by evolving finite-state machines (FSM) for symbol string transformations. In contrast to evolutionary strategies the selection of individuals in EP is typically stochastic rather than deterministic and there is usually no recombination (crossover) operator. The representation of the individuals in evolutionary programming is problem domain dependent and can involve ordered lists, graphs, and for most optimisation problems real values.

The probably most popular evolutionary computation concept for parameter optimisation however has been genetic algorithms pioneered by Holland [42] and developed further by De Jong [20]. The success of this concept is primarily due to the representation of the genotypes, which are strings of genes[15]. In early work this representation involved fixed-length binary strings and later also strings of variable length [36]. In contrast to evolutionary strategies and evolutionary programming typically operating on the phenotypic level this string representation of the genotypes is problem domain independent. This domain independence is provided since a GA practically operates on standardised meta-information (bit strings) about the solution. A substantial body of theory has been established around this representation and its

[15] In GA context genes are primitive components of a solution. In early work [42] individuals were encoded as fixed length binary strings, where bits represented genes. In other work strings of real valued numbers were used.

benefits. The Schema Theorem, Implicit Parallelism [42] and Building Block Hypothesis [35] have particularly contributed to the popularity of genetic algorithms.

An evolutionary paradigm that has attracted much attention recently is genetic programming (GP). While evolutionary programming, evolutionary strategies and genetic algorithms usually involve optimising parameters of a solution, in genetic programming computer programs are evolved to solve a particular task. The idea of evolving computer programs was originally proposed by Cramer [17], but the theory and standards were established by Koza [49].

Beside these main paradigms many alternative evolutionary computation methods have been developed such as Classifier Systems, LS systems etc. An overview of different methods with a vast number of references can be found in [8].

3.2 The evolutionary algorithm

The typical generational evolutionary algorithm is shown in Figure 3-1 using abstract syntax.

```
P:=Create_initial_population()
Measure_Fitness(P)
generation_index:=0
while (not terminated)
{   M:=∅
    repeat
        S:=select_parents(P)
        recombine(S)
        mutate(S)
        M:=M∪ elements(S,|P|-|M|)
    until (|M|=|P|)
    P:=M
    Measure_Fitness(P)
    increment(generation_index)
}
```

Figure 3-1: Generational evolutionary algorithm

The algorithm starts with the initialisation of a population **P** of individuals. This process may be random, or biased if initial knowledge about the solution is available. Then a fitness measure is carried out on all individuals in the population. This usually involves decoding of the genotypes (data structures of the individuals) into

their corresponding phenotypic representation in the problem domain, where the fitness measure is eventually taken.

Before creating a new generation the population **M** receiving offspring individuals is initialised as an empty set. In order to breed a new population the sequence of selection, recombination and mutation has to be iterated until the new population is entirely filled with offspring. In each iteration a set **S** of individuals (often called a *mating pool*; consisting usually of 2 parent individuals) from the old population **P** is selected based on the fitness of the individuals. To generate new offspring the parent individuals in set **S** may then be subject to recombination (mating), where genotypic material is swapped between individuals e.g. by applying the *crossover* operator. Subsequently, mutation may be applied to some individuals in **S** by randomly modifying their genotypes. Both, recombination and mutation may be performed with a certain probability (crossover rate, mutation rate). If neither evolutionary operation was applied to any individual in **S** this particular iteration resembles a plain reproduction cycle where the children are identical copies of their parents. Finally, the generated offspring (typically one or two children) in the updated set **S** will be added to the new population **M**. Since the number of generated children varies between the iterations depending on which evolutionary operator has been used, the last update in a generation may involve more individuals than necessary to saturate the new population. In that case the function *elements* returns only the subset of **S** required to complete the new population by discarding a possible surplus of individuals. This is controlled by the second parameter, which is passed to the function as the number of free slots in the new population calculated by the difference of the cardinalities of old and new population. A generation is completed if this difference is zero. The old population will be replaced with the new one followed by the fitness measure of all individuals in the new population. The evolutionary algorithm proceeds with breeding new populations until a termination criterion is met, typically if a good solution was found or a certain number of generations has been completed.

The traditional generational evolutionary algorithm uses two populations for the parent and offspring generation. A different concept to the generational algorithm is the steady- state algorithm, which involves only one population. The offspring generated by the EA is inserted back into the same population the parents were

picked from by replacing selected, usually poorly performing individuals. This creates a generational overlap of individuals within the population and permits a direct competition of parents with offspring in the evolutionary process. Since the life spans of the individuals during the evolution may be different (some individuals may even survive the whole evolutionary process unchanged) the actual numbering of generations is difficult. Hence the steady-state evolutionary algorithm is said to have completed a generation if the number of children created since the last generation change is equal to the size of the population.

3.2.1 Fitness evaluation

Fitness evaluation of an individual is performed by examining its performance in the problem domain. In implementations of evolutionary algorithms such as evolutionary strategies that optimise a set of model parameters this would typically include the transfer of those parameters into the particular model and subsequently the assessment of the accuracy of that model. For domain independent representations of the genotype used e.g. by GA's the fitness evaluation involves decoding[16] (or interpreting) the genotype of a solution into its phenotypic representation. This process is known as ontogenetic mapping and establishes the representation of the individual in the particular problem domain, where the actual performance measure can be taken by an objective function. The value returned by this function may be a scalar, but is often a vector since several aspects[17] of performance (multi- objective) may need to be recorded.

The result of the performance measure of an individual is used for the determination of a scalar (usually positive) fitness value. For a scalar performance measure the fitness value may be equal to the return value of the objective function. In the case of a multi-objective performance measure the fitness is determined from the vector returned by the objective function by applying scaling or ranking strategies [33]. The fitness value is assigned to an individual and enables the

[16] Decoding is however not always necessary. For parameter optimisation problems the genotype often used by GA's is a string of real values that undergo modification. Since the same representation is used in the problem domain (binary coding of the parameters) there is no decoding required.
[17] For example gradient information may be included as an indication as to how this particular may be improved.

selection mechanism of an EA to perform a comparative performance analysis between the individuals in a population. The preparation of an individual for the fitness evaluation (decoding) and actual performance measure is carried out by the performance and fitness functions, which both need to be implemented by the user.

3.2.2 Selecting individuals

Having assigned the fitness value to all individuals in a population the breeding process of new offspring can be initiated. Each offspring producing cycle begins with the probabilistic (respectively deterministic in evolutionary strategies) selection of parent individuals according to their fitness. The selection operator has a large influence on the convergence properties of the EA. By favouring better performing individuals for breeding this operator applies *selective pressure* of some degree to the population. The higher the selective pressure the higher is the preference for selecting highly fit individuals for the breeding process. Also, high selective pressure usually increases the *convergence rate*, i.e. the pace with which an optimal solution is approached. A strong preference for only highly fit individuals may however prevent the EA from globally exploring the search space of solutions (biased search). As a consequence the search space is narrowed down too early limiting the diversity of individuals in subsequent populations and may hence lead to local *premature convergence* of the population around a sub-optimal solution. A too low selective pressure in turn increases the time used by the EA to find an optimal solution (possible *stagnation* of evolutionary progress). Therefore it is desirable to use a selection operator that applies high selective pressure and preserves the diversity of individuals in a population.

Commonly used selection methods divide into proportionate-based and ordinal-based selection [56]. Proportionate-based strategies select individuals according to their relative fitness. Holland introduced *proportional selection* [42] with the individual having the normalised fitness being the selection probability $p(\mathbf{s_i}) = \frac{f(\mathbf{s_i})}{\sum_{k=1}^{n} f(\mathbf{s_k})}$ for breeding ($f(s)$ is the fitness value of individual s). This selection scheme however is not applicable for negative individual fitnesses or minimisation tasks [7] unless proper scaling is applied. *Roulette Wheel selection* [35] is another popular proportionate selection strategy where each individual is assigned

a slot in the selection interval $[\mathbf{0}, \mathbf{1}]$. The size of such a slot adjusts in proportion to the normalised fitness of the particular individual in the population. The slots in the interval are positioned according to the order of the individuals in the population. The selection is performed by drawing a uniform random number λ, which acts as an "index" pointing into that selection interval and chooses the individual found at this index similarly to a Roulette wheel. Convenient for the implementation is that individuals do not need to be ordered according to their fitness. However, this selection method introduces a bias for highly fit individuals since they receive larger slots in the selection interval. Another problem occurs when the population converges and the differences between fitnesses of the individuals become smaller. The method then resembles random selection with no guarantee of propagating fitter individuals (stagnation). To overcome this problem fitness scaling[18] needs to be introduced which amplifies the differences in fitness between individuals. Baker [9] introduced *stochastic universal sampling*, which works similar to roulette wheel selection, however with no bias and minimal spread[19]. This method chooses *n* individuals for the mating pool simultaneously from the selection interval. *n* equidistant pointers (distance *1/n*) are used as "indices" into the selection interval beginning from the initial "index" given by a uniform random number.

Contrary to proportionate-based methods ordinal-based selection techniques are those, which do not pick individuals based on their fitness value. Instead they introduce a ranking of the individuals within the population based on their fitness and select those individuals according to their position in this ranking. Through ranking the selective pressure is independent from the fitness distribution of the individuals [56]. *Truncation selection* [60] introduces a threshold $\mathbf{T} \in [\mathbf{0}, \mathbf{1}]$, which is used to identify the proportion of the population with the fittest individuals from which parents will be selected. In this proportion all individuals are equally ranked for selection. This method requires the individuals in the population to be ordered according to their fitness. Truncation selection is equivalent to deterministic (μ, λ)-

[18] Scaling is performed by multiplying some constant to all fitness values. This however changes the selective pressure and therewith the statistical properties of the selection scheme. Therefore care needs to be taken in the choice of the scaling method. Contrary to proportionate selection methods ranked based selection techniques are translation and scale invariant [56]. That is neither adding (translating) nor multiplying a constant to all individual fitness values would change the selective pressure of the selection scheme used.

[19] *Spread:* Range of possible values for the number of offspring of an individual [7].

selection used in ES with $\mathbf{T} = \frac{\mu}{\lambda}$ [6] where λ is the number of parents and μ a number of best performing offspring individuals.

Using *ranked selection* all individuals in a population are sorted according to their fitness, and assigned an index describing its rank in the population. This implies that each individual has a different rank and hence usually a different selection probability, even if there are individuals with the same fitness. Since the probability for an individual to be selected does not (directly) depend on its fitness, ranked selection avoids the problems of proportional fitness selection such as stagnation (see proportional selection section 3.2.2) and necessary scaling [35][80]. In *linear ranking* [38] the selection probability is assigned linearly to individuals (more specifically to their ranking index) across the population according to:

$$p_i = \frac{1}{N}\left(2 - \eta^+ + (2\eta^+ - 2)\frac{i-1}{N-1}\right); \qquad i \in \{1, \ldots, N\} \text{ [13]}$$

with $1 \leq \eta^+ \leq 2$ being the *selection bias* to adjust the selection pressure and *N* the population size. The individuals in the population need to be ordered according to their rank (The first individual has the lowest rank and the last the highest). The higher η^+ the higher is the preference for better individuals with the best individual being selected according to the probability $\frac{\eta^+}{N}$. In exponential ranking as one example of *non-linear ranking* the selection probability of a population index (population is ordered as in linear ranking) can be obtained from

$$p_i = \frac{c^{N-i}}{\sum_{j=1}^{N} c^{N-j}}; \qquad i \in \{1, \ldots, N\}$$

with $0 < c < 1$ being the parameter to determine the exponential characteristic of this mapping. A low value for *c* reflects a strong exponential bias for highly fit individuals while a value close to 1 resembles approximately linear ranking. The normalisation by $\sum_{j=1}^{N} c^{N-j}$ guarantees the compliance of the fitness distribution within the population to the uniformity condition $\sum_{i=1}^{N} p_i = 1$. There are however variants of this methods that do not meet this constraint [55].

A very popular and often used method based on ranked selection is *tournament selection*, which is also applied in this research.

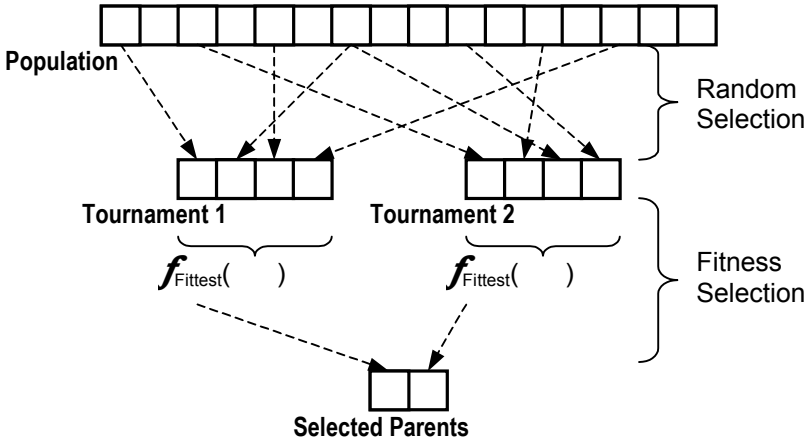


Figure 3-2: Binary tournament selection

This selection technique uses *n* sets of individuals (also known as tournament sets; for $n = 2$ the selection technique is referred to as binary tournament selection shown in Figure 3-2). These sets are filled with a number (typically smaller than the size of the parent population) of uniformly randomly picked individuals from the parent population. From all tournaments the fittest individuals are then chosen into the mating pool for breeding.

The popularity of tournament selection is due to its computational efficiency and statistical properties [53]. Due to the independent random selection[20] of individuals for the tournament sets the parent population does not need to be ordered (no pre-processing of the population is required) which makes this selection method fast. By altering the tournament size the selective pressure can be varied. A low tournament size corresponds to a low selective pressure. In fact, a tournament size of 1 would result in uniformly random selection while a large tournament size allows more individuals from the parent population to be compared to find the fittest and hence increases the selective pressure.

A general issue in evolutionary computation is the potential loss of good genotypes. This can be prevented by applying *elitist* selection, which carries the fittest parent unchanged over to the offspring generation if this individual has not yet been copied by reproduction. Elitist selection was introduced in GA context by De

[20] Independent selection, individuals may be picked more than once into a tournament.

Jong [20] and found to improve the performance of the GA particularly in uni-modal but to degrade the performance in multi-modal search spaces.

3.2.3 Generating offspring

After selecting parent individuals the next breeding step involves the actual creation of offspring. There are in general three ways of producing new individuals from parent individuals: recombination, mutation and reproduction.

Recombination is often used and involves two or more parents swapping genotypic material (sexual reproduction) by applying a crossover operator, which needs to be adapted to the particular representation of the genotype. This is the only offspring-generating operator that actually benefits from the populational search concept in EC in the sense that it combines components of different good performing solutions from the population to produce a new solution. In order to apply crossover the genotypes need to provide a certain number of crossover points, places at which the solution can be broken down into their constituent parts. Crossover can be applied at a single point, i.e. each parent only swaps one component. Alternatively, with multi-point crossover (this is demonstrated by De Jong [21]) an individual may swap several components simultaneously with other parents. However the recombination operator should always adhere to a closure property, which means that it should always produce valid offspring. Invalid offspring would cause the fitness evaluation to fail and therefore make the fitness assessment of those individuals impossible.

Introduced and promoted by Holland [42] recombination builds upon a significant body of theory in GA context. As indicated in section 3.1.1 the genotypes in GA are gene strings, which appear to be a representation ideally suited for crossover. Holland showed that by swapping string fragments of individuals during the course of evolution string patterns or schemata accountable for fitness emerge in individuals across the population. It is believed that fit individuals contain a set of schemata that this fitness can be attributed to. Since individuals are selected according to their fitness beneficial patterns are combined and passed on to the offspring resulting in potentially better performing solutions. The existence of several different schemata in an individual involves many of these patterns when undergoing modification e.g.

by crossover which is known as implicit parallelism and an explanation for the efficiency of genetic algorithms established by Holland in his *Schema Theorem* [42]. Goldberg explained in his *Building Block Hypothesis* [35] the efficiency of GA's by attributing the fitness of individuals to the presence of certain string sequences (building blocks) in the genotype. Using crossover building blocks from different individuals can be combined to form fitter offspring and thus spread across future populations.

In contrast to recombination the generation of offspring by *mutation* involves only one parent individual (asexual reproduction), hence there is no interaction between different genotypic material in the population. The populational search concept of evolutionary computation is only exploited through the wide coverage of the search space by the diversity of individuals. During mutation a genotype undergoes random modification to some degree depending on the representation. In parameter optimisation problems with binary representation of the individuals this usually involves flipping one or more bits. In genetic programming on the other hand the mutation operator is very complex (see section 3.3.4). Mutation is particularly useful in cases where certain genotypic material is needed, which is not contained in any individual in the population. During the evolution it may happen for example that individuals carrying certain genotypic material extinct. In a different context however, these particular genotypic characteristics may greatly improve the performance of some individuals in subsequent populations and might therefore be restored by mutation.

Contrary to recombination in GA's, mutation has been advocated by Fogel [31] who used it as the only offspring-generating operator in evolutionary programming (EP). Similarly, early work in evolutionary strategies (ES) [65] involved only mutation for breeding. Both, EP and ES differ in the application of crossover, which is deterministic in ES and stochastic in EP.

Finally, *reproduction* is the plain generation of copies of the parent individuals, which is useful if the parents have a high fitness e.g. above the population average. Reproduction is also often performed in recombination cases where generated offspring is less fit than the parents or even invalid, which e.g. occurs at times in genetic programming if tree generation constraints are violated for example as a consequence of code bloat [53].

3.3 Genetic programming

One of the main differences between evolutionary computation paradigms is the representation of the individual solution undergoing modification. Chosen depending on the problem context this representation in conjunction with appropriate evolutionary operators provide a certain degree of freedom for finding a good performing solution. In classical EP, ES and GA these representations involved sets of parameters hence constraining the search to the space of possible parameter configurations. Genetic programming (GP) departs from these approaches of numerical search by attempting to evolve symbolic computer programs that solve the problem. Hence GP can be viewed as a stochastic program compilation method. Computer programs are very complex (usually variable-length) compositions of program code and data. Thus this representation spans a large search space of possible programs (there are usually several alternative algorithms) for the evolutionary algorithm to be explored. However, this large degree of freedom also explains the potential GP has to find or to program very good performing solutions[21].

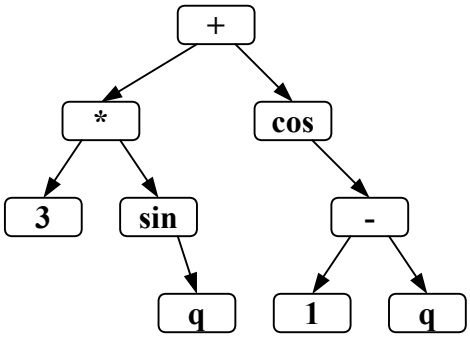
The general idea underlying genetic programming namely the evolutionary *synthesis* (or *induction*) of computer programs from primitive components has been initially investigated by Cramer [17] who designed a "number string" language (JB and TB language) to code simple sequential programs that undergo adaptation by a genetic algorithm. The main principles of genetic programming and its terminology based on the LISP programming language were introduced by Koza [49].

3.3.1 Representation of individuals

Since it involves the synthesis and evaluation of computer programs GP works on the same level as an interpreter for programming languages and hence utilises program and data structure concepts used by those tools. To implement a GP system Koza [49] used the LISP programming language since it provides built-in mechanisms to modify program code and data syntactically in the same way at

[21] There are however concepts e.g. for program modularisation such as ADF (Automatic Defined Functions, see e.g. [50][66]) which can greatly improve the search efficiency in domains where several similar tasks have to be learned. Those tasks could be described with one program module being called with different arguments (see e.g. the lawnmower example in [50]).

runtime. He called the evolved programs *s-expressions*, which are internally represented by rooted labelled *trees* containing two kinds of nodes: inner tree nodes or *non-terminal nodes* and leaf nodes or *terminal nodes*.



$$\mathbf{3} \cdot \sin(\mathbf{q}) + \cos(\mathbf{1} - \mathbf{q})$$

Figure 3-3: Genetic programming tree example for a mathematical expression

Non-terminals have at least one argument (child node or branch) that is there is at least one arc leaving the node (out-degree or *arity* > 0). *Terminal* nodes are atomic nodes (no arguments are to be evaluated) with no leaving arc (out-degree = 0). All genetic programming trees have a root from which the evaluation starts. The evaluation is performed depth-first and typically by evaluating children from left to right. The depth of a tree is the maximum number of arcs on the way from the root node to the most remote leaf node. Terminals therefore have a depth of zero while function trees have a depth equal to or greater than zero.

Non-terminal and terminal sets are the only resources of elementary node definitions a GP algorithm uses to compose programs. Both sets are defined in problem context. The non-terminal set may contain arithmetic, trigonometric, Boolean functions, conditional operators (if-then-else) and other problem domain depended functions such as motion commands etc. The terminal set typically contains constants, variables or other primitive problem domain-dependent entities such as functions or commands without arguments. With symbolic regression the terminal set may also contain an ephemeral random constant $\Re$, which returns a random number in a specified range every time this terminal is chosen. An important requirement for non-terminal and terminal set is *sufficiency*. This means both sets need to have all node definitions that are necessary to induce a program that solves

the particular problem[22]. Also, all nodes in particular function nodes are required to adhere to the *closure* property, i.e. the evaluation result of any node is always well defined. Since GP is typically unconstrained[23] in the way of constructing programs, the closure property ensures that all nodes (terminals and non-terminals) have the same type. Also, possible invalid operations need to be prevented by securing (redefining) all used operators that may have undefined results in certain situations. For example if the division operator is used, the protected division[24] $a\%b = \begin{cases} 1 & ;\mathrm{b}{=}0 \\ a/b & ;\mathrm{else} \end{cases}$ needs to be introduced to prevent invalid results from a possible division by zero. Correspondingly the protected square root is defined as $\mathrm{sqrt_p}(x) = \sqrt{|x|}$ or the protected logarithm as $\log_p(x) = \begin{cases} 0 & ;\ \mathrm{x}{=}0 \\ \log(|x|) & ;\ \mathrm{else} \end{cases}$.

3.3.2 Tree generation

Individual trees can be created by two recursive methods: GROW and FULL [49]. The GROW algorithm (Figure 3-4) generates trees in which leaf nodes may have different distances (number of arcs) to the root node. Supplied with the maximum tree depth parameter **D,** the algorithm first generates a local root node. If the maximum tree depth is reached this node will be a randomly chosen terminal node since no further tree growth from this local root is permitted. If the maximum tree depth is not reached yet this node is randomly chosen from the union set of terminals or non-terminals. If a terminal was chosen the tree growth ends at this node. If a non-terminal was chosen the algorithm recursively creates its arguments trees and thus increases the depth of this particular branch.

[22] *Sufficiency* in genetic programming is a quite similar concept to *completeness* in context of parametric kinematic modelling (see section 2.2). Both, sufficiency and completeness are requirements to be met (typically by human intervention) in order for the respective paradigm to succeed. Whereas completeness is required to numerically identify true parameter values, sufficiency is essential to enable the GP algorithm to find the actual model structure and parameter values.

[23] However, usually there is a maximum tree depth constraint that prevents the tree offspring from growing beyond a certain limit (see also section 3.3.4). Another example: Since in strongly typed genetic programming [58] trees may have different types, a constraint has been introduced that ensures type compatibility between formal parameters in the function declaration (non-terminal set) and actual arguments.

[24] It is important to preserve the precedence level of the protected division operator when representing symbolic expressions in infix notation without redundant parentheses. For example, using the ordinary division operator the expression $a * (b/c)$ is equivalent to $a * b/c$. However, $a * (b\%c)$ is, due to the definition of the protected division (see above), not equivalent to $a * b\%c$!

```
function GROW(D,T,N) //D=Tree depth; T=Terminals; N=Non-terminals
{  if (D=0) then
                return randomly chosen terminal from T;
   else
   {  t := randomly chosen from  T∪N;
      if (t is a terminal) then return t;
      else
      { Instantiate each parameter in t by GROW(D-1,T,N);
        return t with instantiated parameters;
      }
   }
}
```

Figure 3-4: The GROW algorithm

The FULL algorithm always creates balanced trees (see the left-hand parent shown in Figure 3-6), which means that all leaf nodes in a tree have the same distance to the root node. While trees generated by the GROW method using the same depth parameter **D** may vary in depth across the range of integers $\{0, \ldots, D\}$, the FULL method always produces trees with a depth specified by **D**.

```
function FULL(D,T,N) //D=Tree depth; T=Terminals; N=Non-terminals
{  if (D=0) then
                return randomly chosen terminal from T;
   else
   {  t := randomly chosen non-terminal from N;
      Instantiate each parameter in t by FULL(D-1,T,N);
      return t with instantiated parameters;
   }
}
```

Figure 3-5: The FULL algorithm

These tree generation algorithms are an important part of a GP system for creating the initial generation of individuals and for the subtree mutation operator (see section 3.3.4).

3.3.3 Initialisation of the first population

Before the evolution begins an initial population needs to be created. To provide a large diversity of programs the most widely used method to generate the first population is RAMPED HALF & HALF proposed by Koza [49]. This method combines GROW and FULL method and is capable of generating trees of various shapes and depths. It splits the population into intervals in which one half of the individuals are created by the GROW method and the other by the FULL method. Each interval represents a different tree depth starting from the minimum to the maximum tree depth forming a tree depth ramp. The minimum tree depth is usually greater than zero in order to prevent terminal nodes from becoming individuals (zero-depth individual) in a population. Terminal nodes only have one crossover point (respectively mutation point). Hence single terminal nodes being individuals in the population usually have less potential to introduce larger changes to their offspring. In fact, the crossover operator is disabled for two zero-depth individuals. In this scenario the parents are only swapped and reproduced without any changes to the genotypic material. The only way to produce offspring from zero-depth individuals that is different from its parents is the application of mutation (e.g. by changing variable values or replacing the terminal nodes by randomly generated trees). Since the GROW method may occasionally produce zero-depth trees (plain selection of terminals), those trees need to be discarded. The GROW method is then applied with the same parameters until a tree with a desired minimum depth> 0 has been created.

3.3.4 Crossover and mutation

In genetic programming offspring is generated by applying crossover and mutation[25] to selected parent individuals.

Subtree crossover as shown in Figure 3-6 is performed by selecting one crossover node (also known as single point crossover) in each of both parents and swapping the tree fragments rooted at these nodes. Since the crossover operator may select subtrees of different depths, the offspring trees may grow. Larger trees imply

[25] Initially crossover was the only offspring-producing operator in GP [49].

expensive evaluation. They are however not necessarily fitter than less complex trees, which is one reason why GP systems impose a maximal depth constraint on the trees in a population. If a tree generated by crossover exceeds this limit it will be discarded. If the crossover operator fails to generate valid offspring after a certain number of repeated attempts, the parents will be reproduced instead[26].

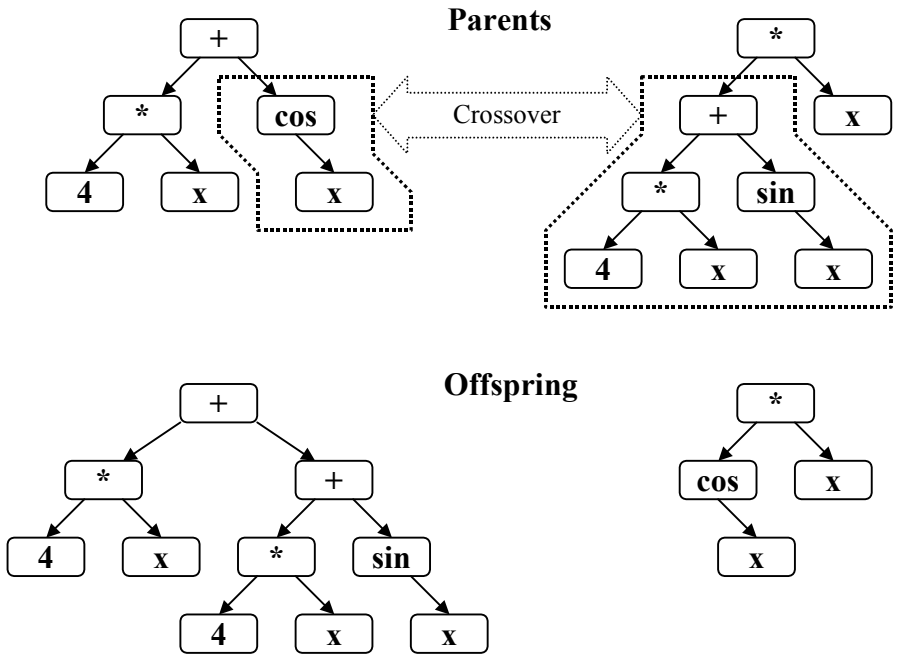


Figure 3-6: Subtree crossover example

Since swapping non-terminal trees usually results in larger performance changes[27] between parents and offspring, the probability of selecting non-terminals for crossover is typically 90% [53].

The mutation operator selects a node in a parent tree and replaces the tree rooted at this node by a randomly generated tree (See Figure 3-7). This new subtree may be produced by either FULL or GROW method with the depth parameter being $\mathbf{D} = \mathbf{D}_{\mathrm{Max}} - \mathbf{D}_{\mathrm{MP}}$, where $\mathbf{D}_{\max}$ is the maximal depth allowed for trees and $\mathbf{D}_{\mathrm{MP}}$ is the depth of the tree rooted at the mutation point. This ensures that the mutated tree

[26] This phenomenon occurs more often in later generations as the trees usually grow during the course of evolution. This code bloat however presents serious problems for the evolutionary progress as the applicability of crossover as the main offspring-generating operator becomes limited or even ineffective as more parents are reproduced [53].
[27] This is a reason why genetic programming is less sensitive to premature convergence than genetic algorithms.

will always be of valid length. Mutating non-terminal nodes is referred to as macro mutation (The same concept applies if fit individuals swap trees with random individuals [63]) whereas micro mutation involves terminal nodes. Shrink-mutation is a process where a subtree is replaced with another tree having a lower depth eventually leading to a lower depth of the overall tree. An example for grow-mutation where the modification of the individual results in a higher tree depth is depicted in Figure 3-7. Other forms of mutation implement swapping subtrees within an individual, replacing trees with trees of the same arity (number of arguments), modifying random constants etc. (see [16] for reference).

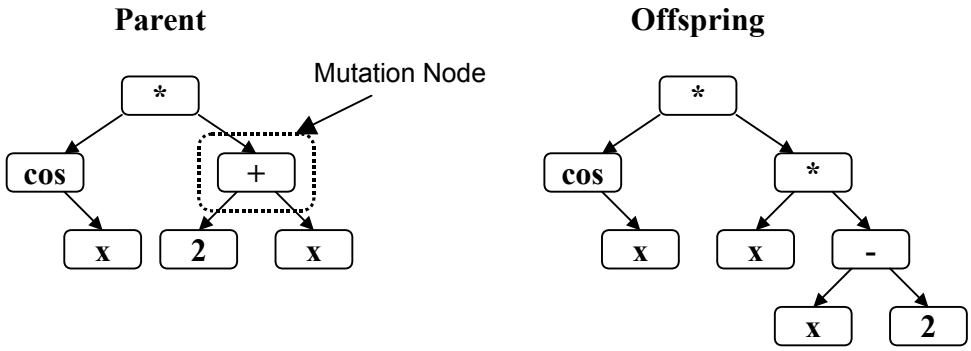


Figure 3-7: Example of subtree mutation in GP

3.3.5 Symbolic regression

Genetic programming has been applied to a variety of problem domains such as structured process modelling [64], parameter identification (e.g. analysis of the existence and identification of multi-steady states in non-linear dynamical systems [57]), machine learning (e.g. evolution of multiplexers; control strategies of autonomous robots [61][28]), artificial intelligence (e.g. synthesis of artificial neural networks by grammar evolution [39], navigation strategies e.g. of an artificial ant in the "Santa Fe Trail" [49] or of a Lawnmower [50]) etc[29].

[28] Interestingly, in this approach genetic operators are applied to linear strings (rather than trees; → Linear Genetic Programming) of 32 Bit CPU instructions. Since machine code is generated the concept is called Compiling Genetic Programming (CGP).
[29] See also [51] for further applications of genetic programming.

The GP domain addressed in this work is symbolic regression [49]. Classical mathematical regression techniques (traditional robot calibration methods belong to this category) typically utilise a regression model (e.g. linear, non-linear, parametric) pre-specified by a user according to the requirements of the problem domain. The parameters of these models are subsequently fitted to measured data. Symbolic regression in contrast attempts to find (or induce) a symbolic description for the relationship typically between two variables (action and response) by evolving an appropriate mathematical model consisting of functions and parameters (see example application in [48]). Since symbolic regression is not limited to a predefined model structure it has more potential in terms of accurate modelling than conventional regression methods. In fact, it is principally capable of identifying the true functional relationship of variables based on measurement data, provided sufficiency of terminal and non-terminal set is given. The terminal set must contain the dependent variables[30] and usually an ephemeral random constant. The non-terminal set contains mathematical functions and operators that are assumed to be part of the solution, for example, periodic data is likely to be modelled by sine/cosine-functions. The principal representation of the mathematical expressions that are subject to evolution is shown in Figure 3-3, Figure 3-6 and Figure 3-7.

Fitness measures typically applied in genetic programming and particularly in symbolic regression are raw fitness, standardised fitness, adjusted fitness and normalised fitness [49]. A fitness measure often used (also in this work) is raw fitness $r(i) = \sum_{j=0}^{N} |S(i,j) - C(j)|$ of the i^{th} individual with $S(i,j)$ being the value returned by the individual and $C(j)$ the value to be pursued for the j^{th} out of N fitness case. Standardised fitness is defined either as $s(i) = r(i)$ or $s(i) = r_{max} - r(i)$, adjusted fitness as $a(i) = \frac{1}{1+s(i)}$ and normalised fitness as $n(i) = \frac{a(i)}{\sum_{k=1}^{N} a(k)}$.

[30] Variables used in symbolic regression are usually continuous. However, function induction is also possible in discrete domains (multiplexer synthesis using Boolean variables).

3.4 Summary

This chapter introduced the principles of evolutionary computation as alternative stochastic search methods to conventional deterministic optimisation techniques. Based on [32] the general advantages of EC techniques are summarised as:

- Domain independence and hence a wide area of applications (The same algorithm (Figure 3-1) applies to all implementations (conceptual simplicity) with specially adapted fitness function and evolutionary operators).
- Incorporation of domain specific knowledge possible.
- Parallelism (The process of breeding new individuals is an independent local process and can be performed in parallel).
- Good chances for global convergence (Usually outperforms gradient methods in multi-modal search spaces due to better search space coverage by multi-individual search).
- Reuse of solutions (Exploiting good characteristics of former solutions).
- Potential to solve problems with unknown solutions or in areas with no human expertise

If solutions to problems can be formulated as computer programmes the performance of which can be evaluated by computers, genetic programming can be applied to populations of these solutions to automatically combine or modify their different subprograms to eventually form optimal programs.

Section 3.3 presented genetic programming, which is the evolutionary paradigm used in this work. The principles of genetic programming were introduced with focus on symbolic regression, which is the application domain for the inverse kinematic calibration method described in Chapter 4.

Chapter 4

Static symbolic robot calibration based on genetic programming

This chapter presents a new application of genetic programming particularly symbolic regression to static inverse kinematic robot calibration in context of offline programming. Building upon the robot calibration and evolutionary computation basics introduced in chapters 2 and 3 a general symbolic calibration system is developed and its principles and properties detailed.

4.1 Evolutionary calibration concept

As outlined in section 2.1.3 the objective of static calibration in context of offline programming is to establish an accurate calibration model, which provides a mapping between nominal and corrected end-effector poses. These corrected poses represent false targets that compensate for path deviation and eventually lead the robot to the desired location. This pose-correcting mapping can either be based on parametric or non-parametric models. The principle structure of the mapping model used in this work is illustrated in Figure 4-1. This model fits into the category of compensation models that use a nominal inverse kinematic model as outlined in section 2.2.4 and illustrated in Figure 2-6. Hence it avoids the problems of finding the calibrated inverse of the overall kinematic model (section 2.2.4), which expresses the calibrated joint configurations indirectly by calibrated parameters in geometric

and non-geometric model components. The calibration model in Figure 2-6 is represented by a vector of joint correction functions followed by a nominal forward kinematic model. The joint correction functions, which are subject to calibration, describe herein the error of the nominal inverse kinematic model (as used by the robot controller) hence it fits into the category of inverse kinematic calibration. This calibration principle using correction functions was initially tested by Shamma [70] (recently by Jenkinson [45]) and is outlined in section 2.2.4. However, in Shamma's approach pre-specified polynomial functions have been used with little of no physical significance. The accuracy of those models depends on the functions used and is hence also pre-specified. The calibration of these models again involved numerical identification of the parameters (coefficients).

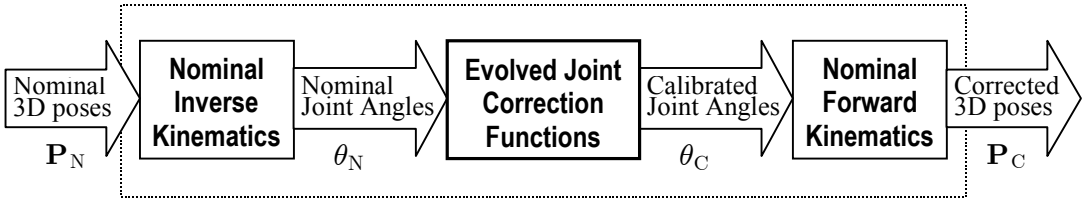


Figure 4-1: Calibration model: Joint correction functions are evolved in context of nominal inverse and nominal forward kinematic models establishing a mapping between nominal and corrected 3D poses

In this research the joint correction functions are evolved by applying genetic programming particularly symbolic regression to the calibration problem. In result correction functions for joints are obtained in symbolic form rather than numerical values of parameters within predefined models. Therefore this calibration method is denoted as symbolic calibration. By providing appropriate sets of terminals and non-terminals (e.g. functions that typically occur in non-geometric models; see e.g. section 2.2.2) this approach has the potential not only to compensate for positional error of the manipulator, but also to identify the structure and parameters of the correction functions enabling further mathematical analysis.

Performing the calibration on joint level the method primarily avoids the problems related to task space correction of redundant manipulators. A redundant manipulator can reach a pose using different joint configurations. Since the positional error caused by different joint configurations is typically different, the calibration algorithm would have to calibrate a one-to-many mapping. This would

result in an averaged compensation for these different joint configurations, which does not reflect the error in the individual configuration. This problem does not arise when calibrating joint correction models as proposed in this work. Possible different joint configurations of a pose are obtained as different solutions from a nominal inverse kinematic model, which is not subject to calibration.

4.2 Pose correction principle

Prior to the actual calibration of the model in Figure 4-1 a set of data pairs $(\mathbf{P}_N, \mathbf{P}_C)$ has to be generated where $\mathbf{P}_N$ is a nominal pose e.g. generated by an OLP system, and $\mathbf{P}_C$ the corresponding corrected pose (false target) to compensate for positional error and lead the robot to the desired pose. The robot is programmed to move to the nominal positions and the actual achieved positions are recorded by an external measurement device. In the experiments reported in this work the Robotrak measurement system was used (See section A.1 and A.1.1 for reference about measurement principle and data transformations). A corrected pose is obtained by subtracting the positional error represented by the difference between corresponding measured pose $\mathbf{P}_M$ and nominal pose $\mathbf{P}_N$ from the nominal pose (see Figure 4-2). Note that in this work only the position vector of a pose is subject to correction.

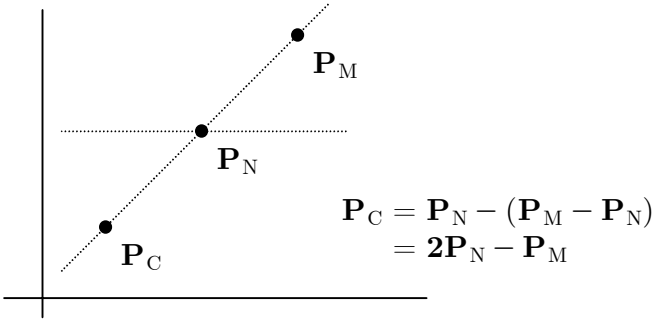


Figure 4-2: Illustration of the correction principle

Corrected poses for the calibration data set are established from nominal calibration poses using the algorithm shown in (Figure 4-3). Arguments for this algorithm are nominal end-effector poses and local frame poses[31]. First the robot is

[31] In this work calibration is carried out relative to a local frame (see also [45]). The local frame concept is explained in the appendix section A.1.1.

sent to all local frame points and the end-effector poses are measured and stored in $\mathbf{L}_{LM}$. Then the robot is sent to all poses in the calibration data sample set **P** and the corresponding end-effector poses are measured and stored in $\mathbf{P}_{LM}$. Since the calibration poses have been measured relatively to the measurement system frame, they need to be transformed into robot base frame co-ordinates. The therefore required transformation matrices are constructed from the local frame points expressed relative to robot base frame (set **L)** and measurement system frame (set $\mathbf{L}_{LM}$) (See also appendix section A.1.1). Having established this transformation, nominal and actual poses can be compared and the corrected poses generated as illustrated in Figure 4-2. A further result of the data preparation procedure is an error statistic computed from transformed measured and nominal poses (describing the error prior to calibration), and a set of joint angles generated from the corresponding nominal poses.

```
function data_preparation(P, L)
//P= Robot Poses; L= Local Frame Poses
{  L_LM :=goto_poses_and_measure(L);
   P_LM :=goto_poses_and_measure(P);
   M    :=transform_calibration_data(P_LM, L, L_LM);
   C    :=compute_corrected_poses(M, P)
   S    :=write_error_statistic(M, P);
   J    :=compute_joint_angles(P);
   return {C, J, S}
}
```

Figure 4-3: Abstract data preparation algorithm

4.3 Calibration principle: Distal Supervised Learning

The problem of finding the joint correction functions in Figure 4-1 can be interpreted as a *distal supervised learning problem*. The problem of distal supervised learning is generally stated as finding a mapping from *intentions* to desired *distal results* or *tasks* (Figure 4-4). A model to provide this mapping is trained with pairs of (intention, result) in a supervised fashion. The model internal problem is to find a

mapping that transforms given intentions into appropriate *proximal* actions[32] (inverse mapping), which can then be transformed into the desired *distal* results in the problem domain [46].

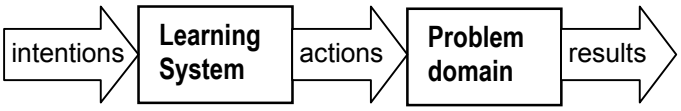


Figure 4-4: Distal supervised learning

To transfer these concepts into the symbolic kinematic calibration domain of this research: intentions are interpreted as nominal end-effector poses $\mathbf{P}_{\mathrm{N}}$, actions as calibrated joint configurations θ_{C} and results as the corrected end-effector poses $\mathbf{P}_{\mathrm{C}}$. Hence the task of the model in Figure 4-1 is to find an accurate mapping between nominal and corrected end-effector poses by establishing the required calibrated joint configurations. Learning of these calibrated[33] joint configurations from nominal joint configurations is performed by evolving joint correction functions implicitly using a forward kinematic (geometric) model as a distal teacher. Implicit learning refers to the fact that the calibrated joint configurations as being the target configurations are not explicitly provided. Instead the evolution of the correction models is driven by their distal performance being the accuracy of the whole forward kinematic model (geometric model + evolved joint correction models).

Correction of a joint configuration θ is performed as shown in equation 4.1 by adding a vector of correction values computed at θ by correction terms stored in $\mathbf{g}$:

$$\mathbf{h}(\theta) = \theta + \mathbf{g}(\theta) \tag{4.1}$$

where $\mathbf{h}$ is a function vector consisting of k joint correction functions where k is the number of joints in θ. The correction term to each function in $\mathbf{h}$ is stored in the corresponding component of function vector $\mathbf{g}$. These correction terms in $\mathbf{g}$ are the actual subject to evolutionary refinement by symbolic regression (In the following the correction terms in $\mathbf{g}$ are also referred to as correction models). At any

[32] There may however be several different proximal actions that have the same distal result [46]. In robot calibration context this is equivalent to the problem of redundant manipulators, which can establish certain poses with different joint configurations.

[33] A calibrated joint configuration θ_{C} results exactly in the desired target end-effector pose $\mathbf{P}_{\mathrm{C}}$. Note however that θ_{C} is not explicitly given in this approach.

evolutionary step the components of $\mathbf{g}$ contain the currently best performing evolved correction model (see section 4.4.1). Using these currently evolved correction models in $\mathbf{g}$ an evolved joint configuration θ_E (as an intermediate result of the evolution from nominal to calibrated joint configurations) is computed from a nominal end-effector pose $\mathbf{P}_N$ as the result of the correction applied at the current stage of evolution:

$$\theta_E := \mathbf{h}(\theta_N) \tag{4.2}$$

$$\text{with} \quad \theta_N := \mathbf{f}^{-1}(\mathbf{P}_N, \phi_N). \tag{4.3}$$

The nominal joint configuration θ_N, which is corrected by $\mathbf{h}$ resulting in θ_E, is obtained from $\mathbf{P}_N$ using the nominal inverse kinematic model (equation 4.3). From the evolved joint configuration θ_E the corresponding evolved end-effector position $\mathbf{P}_E$ can then be obtained using the nominal geometric forward model (in this work a Denavit-Hartenberg model is used; see Appendix section A.2) based on nominal geometric parameters ϕ_N of a Unimation PUMA 761 industrial robot (Table A-1):

$$\mathbf{P}_E := \mathbf{f}(\theta_\mathbf{E}, \phi_N).$$

Hence, using equation 4.2 and 4.3 the entire mapping from nominal to evolved end-effector positions can then be written as:

$$\begin{aligned} \mathbf{P}_E &= \mathbf{f}(\mathbf{h}(\mathbf{f}^{-1}(\mathbf{P}_N, \phi_N)), \phi_N) \\ &= \mathbf{F}_{map}(\mathbf{P}_N). \end{aligned} \tag{4.4}$$

In order to illustrate the location of the correction functions in $\mathbf{h}$ being the arguments of the forward kinematic model the mapping can also be formulated using the sequential structure of the Denavit-Hartenberg parameterisation as:

$$\mathbf{P}_E = \prod_{\mathbf{i=1}}^{\mathbf{k}} \mathbf{A}_i(\mathbf{h}_i(\theta_N)) \tag{4.5}$$

where $\mathbf{k}$ is the number of links, $\mathbf{h}_i$ is the i^{th} correction function (θ_N has been obtained from $\mathbf{P}_N$ using the nominal inverse using equation 4.3) being the argument of $\mathbf{A}_i$ which is the DH transformation matrix (see Appendix A.2) for the i^{th} link ($\mathbf{A}_i$ absorbs the (nominal) DH parameter $\alpha_i, \mathbf{a}_i$ and $\mathbf{d}_i$, hence this formulation is only valid for manipulators with exclusively revolute joints as for the PUMA 761).

During the learning (or training) phase the mapping model (equation 4.4) is presented a number of pairs of nominal and corrected end-effector poses $(\mathbf{P}_{\mathrm{N}}, \mathbf{P}_{\mathrm{C}})$. In order to establish an accurate mapping between all $\mathbf{n}$ training pairs in the calibration sample set the objective of the supervised learning process is to minimise the performance index:

$$\sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{e}^{\mathrm{i}\mathrm{T}} \mathbf{e}^{\mathrm{i}} ; \quad \text{where} \quad \begin{aligned} \mathbf{e}^{\mathrm{i}} &= \mathbf{P}^{\mathrm{i}}_{\mathrm{C}} - \mathbf{P}^{\mathrm{i}}_{\mathrm{E}} \\ &= \mathbf{P}^{\mathrm{i}}_{\mathrm{C}} - \mathbf{F}_{\mathrm{map}}(\mathbf{P}^{\mathrm{i}}_{\mathrm{N}}) \end{aligned} . \tag{4.6}$$

(Note that the superscripted i does not represent exponentiation). The minimisation is performed by evolving the joint correction models (using symbolic regression) in $\mathbf{g}$ so as to minimise the differences between corrected end-effector poses from the training set (desired pose) and those computed (by the mapping model) from the corresponding nominal end-effector pose. Practically, the inverse transformations of the nominal poses $\mathbf{P}_{\mathrm{N}}$ by the nominal inverse model in $\mathbf{F}_{\mathrm{map}}$ may not be computed explicitly by the calibration program. Instead the joint configurations of the nominal poses may be taken directly from the robot control system by storing[34] them in controller memory when the robot is sent to these poses to establish the external measurements as part of data preparation algorithm (Figure 4-3). Using calibration (or training) samples $(\theta_{\mathrm{N}}, \mathbf{P}_{\mathrm{C}})$ comprising of nominal joint configuration and corresponding corrected target pose the performance index in equation 4.6 can then be rewritten with:

$$\mathbf{e}^{\mathrm{i}} = \mathbf{P}^{\mathrm{i}}_{\mathrm{C}} - \mathbf{f}(\mathbf{h}(\theta^{\mathrm{i}}_{\mathrm{N}}), \phi_{\mathrm{N}}).$$

The performance index in the experiments reported in this work is a raw fitness measure (see section 3.3.5) used by the EA to select individuals.

4.4 The evolutionary calibration system

The context and the components of the evolutionary calibration system are illustrated in Figure 4-5. The input calibration data for the system are obtained from data preparation module as outlined in section 4.2. For the laboratory PUMA 761 robot used in the experiments the calibration system consists of 6 separate genetic

[34] In VAL II the joint configurations of corresponding Cartesian poses can be logged into system memory as precision points by issuing e.g. “here #p” [72] (see also section 6.1.1).

programming system instances (one for each joint), each of which works on one population of parent models. The fittest correction model of each population in these GP systems at any evolutionary step constitutes the corresponding component in the correction model vector $\mathbf{g}$.

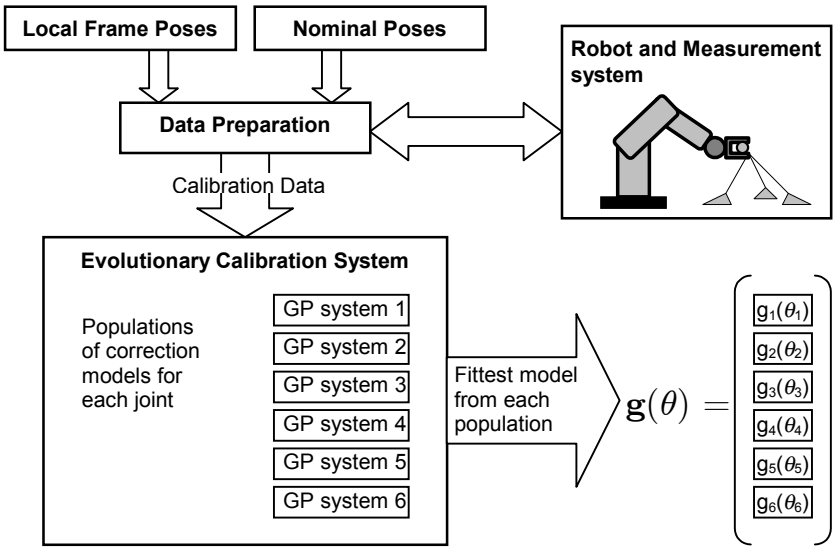


Figure 4-5: Overview of the evolutionary calibration system

4.4.1 The symbolic co-evolutionary calibration algorithm

The principle of the calibration algorithm used to evolve the joint correction models is shown in Figure 4-6. First the populations of correction models are initialised using the RAMPED HALF & HALF method (section 3.3.3). The terminal set contains the joint variable θ and an ephemeral uniform random constant $\Re \in [\mathbf{0}, \mathbf{1}]$. The function set contains functions that typically occur in joint-related non-geometric models i.e. arithmetic functions such as addition, subtraction, multiplication and protected division, and trigonometric functions such as sine and cosine (see section 2.2.2 and section 6.2). At initialisation one non-random neutral

model[35] is introduced to each population. Such a model returns a zero value (no correction), which corresponds to an uncalibrated joint angle. The reason for this is to guarantee (elitism provided) the evolution to start with at least the uncalibrated (nominal) joint configuration. This is particularly important when the algorithm switches to a joint for the first time, which is explained in context later in this section (see page 67).

After creation and initialisation of the populations the components of the correction model vector $\mathbf{g}$ are initialised with neutral models (no correction). In this way it is ensured that the initial performance index (measure of positional error at the beginning of the evolution) of the kinematic model is equal to the performance index of the plain uncalibrated kinematic model (without correction models).

```
k:= Number_of_joints
P:= Create populations of correction models for k joints
for each k do
        g_k(θ) := neutral correction
Compute initial performance index
repeat
    j := Select Joint according to the contribution of its
         corresponding correction model in g to the
         positional error of the end-effector
    Evolve correction model for joint j in population P[j]
         until performance index improves
    Update g_j(θ) := fittest model of population P[j]
until terminated
```

Figure 4-6: Co-evolutionary calibration algorithm

An evolutionary cycle (wrapped into the repeat-until statement in the algorithm illustrated in Figure 4-6) begins by selecting a joint corresponding to the contribution of its correction model in $\mathbf{g}$ to the positional error (see section 4.4.2). As shown in equation 4.5 the kinematic forward model is represented by a sequence of homogenous Denavit-Hartenberg transformations. Hence the correction models in

[35] In the experiments $\mathbf{g}(\theta) = (\theta - \theta)$ (more specifically the right hand side of it) has been inserted as an individual providing no correction into the initial population. Principally, $\mathbf{g}(\theta) = 0$ could have been used. However, the right hand size of this expression is a plain terminal the occurrences of which as single individuals are to be avoided in the population due to their limiting influence on the crossover operator and population diversity as described in section 3.3.3.

their corresponding correction functions (equation 4.1), which are arguments of their corresponding DH transformations, collectively contribute to the performance index within this serial arrangement. This also implies that the correction models depend on each other during the evolutionary learning process. When altering one correction model the other models are directly affected in their potential to minimise the remainder of the positional error. An example is illustrated using a planar manipulator in Figure 4-7:

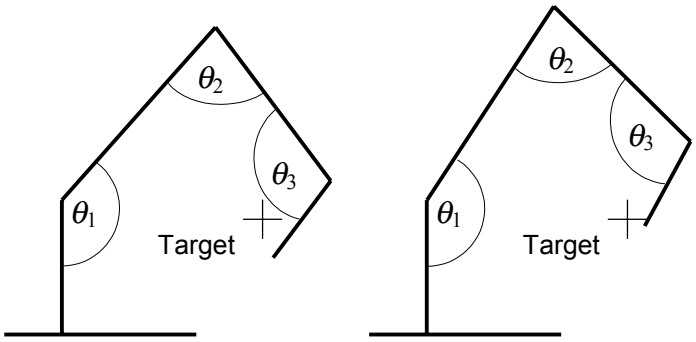


Figure 4-7: Dependence of joint corrections: The positional error between target and tool endpoint was reduced by increasing joint angle θ_1. In effect the correction that needs to be applied to joint θ_3 compared to the previous state is now reduced.

Since this property prohibits a parallel implementation of evolution for each joint (in this calibration method), only correction models for one joint are evolved at the time with the objective to minimise the performance index in the "environment" provided by the fittest correction models of the other joints. This concept is also known as co-evolution since the performance of individuals depends on the performance of individuals in other populations. While one correction model is evolved in its corresponding population, the evolution of the other models is suspended. The generations used by the calibration system (in Figure 4-5) to evolve the correction model vector $\mathbf{g}$ are the sum of the generations completed by each GP system.

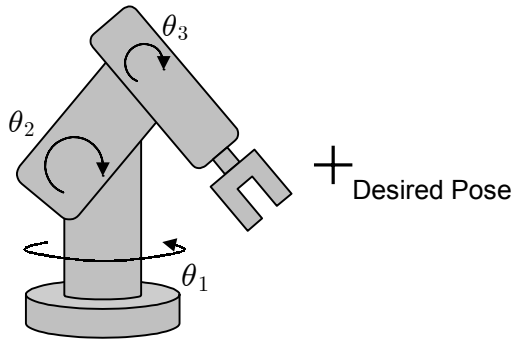


Figure 4-8: Necessity of joint selection: The desired pose cannot be reached by altering joint angle θ_1

The necessity of selecting a correction model for evolutionary improvement depending on its contribution to the positional error is illustrated in Figure 4-8. The depicted 3-link manipulator is not capable of reaching the desired position by modifying the angle of 1st joint since this joint does not contribute to the positional error (The angle determined by X and Y co-ordinates of this pose is assumed to be equal to joint angle θ_1). Hence it would not be possible to minimise the performance index by evolving a correction model for joint 1. Instead appropriate correction needs to be applied to joint 2 and 3 by co-evolving their correction models based on an appropriate joint selection.

A problem occurs when a joint is selected for the first time. Its population of correction models has been randomly initialised with functions the fittest of which however might deteriorate the currently archived performance index (Note that the corresponding component in vector $\mathbf{g}$ (see equation 4.1 on page 61) is updated for evaluation with the fittest model of that population). This would mean a further unnecessary computational expense since the correction model of this joint would have to be evolved until the previous better performance index is regained and the evolution can actually progress. By introducing a neutral model (non-correcting model) to each initial population as explained before it is ensured that the currently achieved performance index of the model (kinematic plus best performing correction functions for all joints so far) is not degraded when the algorithm selects a joint for the first time. Since the GP system used in this work implements elitism, the best performing correction model is carried over to the next generation without changes. Hence the fittest correction model is kept until a better performing model emerges. In

this way it is ensured that the currently achieved correction models and hence the performance index cannot be degraded.

Whenever the performance index (and thus the positional error between evolved and corrected position) has been reduced (which may take several generations) by evolving a particular joint correction model, the next step in the algorithm in Figure 4-6 is to update the corresponding component in the correction model vector **g** with the fittest model of that population. Then the evolutionary cycle starts again by selecting a joint corresponding to the contribution of its correction model in the now updated correction model vector **g** to the remainder of the positional error (see section 4.4.2). This evolutionary cycle is repeated until the performance index drops below a certain limit or the sum of generations used to evolve all joint correction models exceeds a certain number.

Joint number	Gene-ration	Symbolic expression of the fittest correction model	Perform. Index
2	3	theta+SQRTp(0.3988464003418073)*0.0002441480758079775	98.18535
2	4	theta+SQRTp(theta)*0.0002441480758079775	95.77828
2	5	theta+(0.8793908505508591-theta-SIN(theta))*0.0002441480758079775	94.24098
2	7	COS(0.5919370097964416*0.04806665242469558*theta)*theta	88.69122
2	8	theta+theta*(theta-(theta-COS(0.260567033906064))*0.04806665242469558*0.7900631733146153)*0.001244148075807978	88.45305
2	9	theta+0.6468703268532365%(theta-(0.1467635120700705-SIN(0.837916196172979)+theta-(theta-(0.2048097170934172-theta))))*0.0002441480758079775	86.1388
2	10	theta+theta*(theta-0.6468703268532365%(theta-(0.3129062776573992*0.001244148075807978-(theta-0.4824976348155156))))*0.001244148075807978	84.39505
1	14	COS(SQRTp(0.9591051973021638)-SIGN(0.4750205999938963))*theta	83.3114
1	16	COS(SQRTp(SQRTp(SIN((theta+0.03137302774132512)*SIGN(0.03338724936674093))))-SIGN(0.4750205999938963))*theta	65.1133
2	17	theta+theta*(theta-0.6468703268532365%(theta-(0.3988464003418073*0.0002441480758079775-(theta-0.4824976348155156))))*0.001244148075807978	65.11246

Table 4-1: Symbolic expressions of the correction models generated during a typical run of the symbolic calibration algorithm beginning with a performance index of 106.203676 (uncalibrated model without joint corrections)

The results from the beginning of a typical run (captured from the log-file of the calibration system during the experiments) of the calibration procedure is shown in

Table 4-1 which lines up in rows the evolved correction model of a particular joint, the generation[36] at which this correction model emerged and the resulting improved performance index. Based on the calibration data used in this experiment joint 2 is first selected (see details of the selection method in section 4.4.2). Breeding of new correction models in the corresponding (second) GP system (see Figure 4-5) is performed until the performance index could be improved in generation three. The second component in correction model vector $\mathbf{g}$ is updated with the obtained correction model for the second joint. Based on the remainder of the positional error and the updated vector $\mathbf{g}$ joint 2 is selected again for further improvement of its correction model. In this way genetic programming gradually improves the performance index by refining the correction model of the second joint. In generation ten however a correction model emerges which reduces the positional error in a way that gives a correction of joint 1 more potential for a further reduction of the performance index (see section 4.4.2). Hence from generation 11 the algorithm switches to joint 1 and successfully evolves the first correction for this joint in generation 14 with a further refinement in generation 16. Based on the currently evolved correction models for joint 1 and 2 the algorithm switches at generation 16 to joint 1 and proceeds evolving its correction model from generation 17. In this way the calibration algorithm automatically selects and improves joint correction models in $\mathbf{g}$.

4.4.2 Joint selection

A problem of the symbolic calibration method described in section 4.4.1 is that applying corrections to joint configurations inevitably implies an alteration of the tool orientation (applies to the laboratory robot used in the experiments). To approach this problem a joint selection method has been used that selects joint correction models according to their potential to improve the positional error of the end-effector by applying minimal correction. The method is described as follows:

[36] Note that the generations within the single GP systems in Figure 4-5 are numbered globally due to the serial processing. For example, in Table 4-1 after the initialisation the GP system 2 has completed 10 generations. Then from the next generation (globally the 11th) the focus is switched to the GP system for joint 1, which processes locally its first generation.

The overall kinematic forward model can be expressed using the Denavit-Hartenberg parameters as:

$$\mathbf{T} = \begin{bmatrix} \mathbf{Tr} & \mathbf{Tp} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{1} \end{bmatrix} = \mathbf{A_1 A_2 A_3 A_4 A_5 A_6 P}$$

where $\mathbf{A}_{\mathrm{i}}$ is the homogenous DH matrix for the i$^{\mathrm{th}}$ link, $\mathbf{P}$ the 4×4 tool transformation matrix (see also Appendix A.2), $\mathbf{Tr}$ the 3×3 rotation matrix of the tool frame and $\mathbf{Tp}$ the 3×1 position vector (x, y, z) of the tool endpoint.

Provided it is small the positional error contributed to the overall positional error $\Delta\mathbf{Tp}$ of the robot by the ith link is estimated by the total error relation:

$$\Delta\mathbf{Tp}_{\mathrm{i}} = \frac{\partial\mathbf{Tp}}{\partial\theta_{\mathrm{i}}}\Delta\theta_{\mathrm{i}} + \frac{\partial\mathbf{Tp}}{\partial\mathbf{a}_{\mathrm{i}}}\Delta\mathbf{a}_{\mathrm{i}} + \frac{\partial\mathbf{Tp}}{\partial\mathbf{d}_{\mathrm{i}}}\Delta\mathbf{d}_{\mathrm{i}} + \frac{\partial\mathbf{Tp}}{\partial\alpha_{\mathrm{i}}}\Delta\alpha_{\mathrm{i}}$$

where $\Delta\theta_{\mathrm{i}}$, $\Delta\mathbf{a_i}$, $\Delta\mathbf{d}_{\mathrm{i}}$ and $\Delta\alpha_{\mathrm{i}}$ are small errors in the DH parameters. Since the joint compensation algorithm assumes that the positional error is only due to joint variable θ the following heuristic is applied:

$$\Delta\mathbf{Tp}_{\mathrm{i}} = \frac{\partial\mathbf{Tp}}{\partial\theta_{\mathrm{i}}}\Delta\theta_{\mathrm{i}}$$

which sets the error of the geometric parameters a, d and α to zero. For a 6 DOF industrial robot the positional error therefore is

$$\Delta\mathbf{Tp} = \sum_{\mathbf{i=1}}^{\mathbf{6}} \Delta\mathbf{Tp}_{\mathrm{i}} = \mathbf{J}\Delta\theta \tag{4.7}$$

with $\mathbf{J} = [\frac{\partial\mathbf{Tp}}{\partial\theta_1}, \ldots\frac{\partial\mathbf{Tp}}{\partial\theta_6}]$ being the 3×6 positional Jacobian (three positional components and six joints) of $\mathbf{T}$ with respect to the joint variables. This equation relates the joint angle error $\Delta\theta = [\Delta\theta_1, \ldots, \Delta\theta_6]^{\mathrm{T}}$ to the positional error $\Delta\mathbf{Tp}$ of the robot end effector for a particular pose. The potential of a joint to compensate for positional error at a particular pose using only a small change of the joint angle is indicated by the values in the respective column of the Jacobian $\mathbf{J}$ evaluated at the joint configuration corresponding to that pose. High absolute values indicate a high error compensation potential of the particular joint (column index) at the corresponding pose component (row index determines x, y or z value) while low absolute values represent the need for a larger compensation to reduce the error at this pose component. A zero value in $\mathbf{J}$ shows that the corresponding joint is not capable of reducing an error in the particular pose component.

The measure of the potential of joint j to compensate for a particular error described by a data sample (nominal joint configuration θ_{N}, corrected pose $\mathbf{P}_{\mathrm{C}}$; see section 4.3) has been implemented as:

$$\begin{aligned}\mathbf{m}_{\mathrm{j}}(\theta_{\mathrm{N}},\mathbf{P}_{\mathrm{C}}) &:= \left|\frac{\partial \mathbf{Tp}}{\partial \theta_{\mathrm{j}}}(\mathbf{g}_{\mathrm{j}}(\theta_{\mathrm{N}}))^{\mathrm{T}}(\mathbf{P}_{\mathrm{C}}-\mathbf{Tp}(\mathbf{g}_{\mathrm{j}}(\theta_{\mathrm{N}})))\right| \\ &:= \left|\frac{\partial \mathbf{Tp}}{\partial \theta_{\mathrm{j}}}(\theta_{\mathrm{E}})^{\mathrm{T}}\Delta\mathbf{Tp}\right|\end{aligned} \tag{4.8}$$

where $\Delta\mathbf{Tp} = \mathbf{P}_{\mathrm{C}} - \mathbf{Tp}(\theta_{\mathrm{E}}) = \mathbf{P}_{\mathrm{C}} - \mathbf{P}_{\mathrm{E}}$ is the positional error between corrected pose (target) and currently evolved pose[37]. To explain the principle of the performance measure equation 4.8 is rewritten as:

$$\mathbf{m}_{\mathrm{j}}(\theta_{\mathrm{N}},\mathbf{P}_{\mathrm{C}}) := \left|\frac{\partial \mathbf{Tp}_{\mathrm{x}}}{\partial \theta_{\mathrm{j}}}(\theta_{\mathrm{E}})\Delta\mathbf{Tp}_{\mathrm{x}} + \frac{\partial \mathbf{Tp}_{\mathrm{y}}}{\partial \theta_{\mathrm{j}}}(\theta_{\mathrm{E}})\Delta\mathbf{Tp}_{\mathrm{y}} + \frac{\partial \mathbf{Tp}_{\mathrm{z}}}{\partial \theta_{\mathrm{j}}}(\theta_{\mathrm{E}})\Delta\mathbf{Tp}_{\mathrm{z}}\right|. \tag{4.9}$$

The measure is defined as the absolute scalar product of the derivative of the position equation vector with respect to the j[th] joint (j[th] column of the Jacobian) evaluated at the currently evolved joint configuration θ_{E}, and the corresponding positional error vector $\Delta\mathbf{Tp}$. Each of the three terms is the product of a positional error component and the corresponding change of the end-effector when altering the joint angle j by a minimal value. If the absolute value of a term is large it is typically due to a high value of the derivative, which indicates that joint j has a high potential to compensate this particular positional error component. However, the value of a term can be positive or negative. A positive value indicates that the particular error component can be compensated by reducing the value of the joint variable[38]. A negative value of a product term indicates that the value of the joint variable should be increased to compensate for the error component[39]. Different signs of the terms in equation 4.9 indicate that by applying a small correction to the joint variable j the values of the terms would change contrarily. While the (absolute) error in one component is reduced the (absolute) error in other components with opposite sign is

[37] $\mathbf{Tp}(\theta)$ is identical to the notation $\mathbf{f}(\theta,\phi_{\mathrm{N}})$ used in this work for nominal kinematic models since only position measurements were considered.

[38] In this case derivative and error are both either positive or negative. In the first case the positive derivative indicates a compensation of the positive error component by reducing the value of the joint variable. If both factors are negative the negative derivative also indicates reducing the value of the joint variable so as to compensate for the negative error component.

[39] In this case both factors have opposite signs: If the derivative is negative it indicates increasing the value of the joint variable to compensate for the positive error component, as well as a positive derivative does to compensate for the negative error component.

increased. A possible situation is that the terms cancel each other resulting in a zero measure. This means that a small correction (regardless whether positive or negative) applied to joint j would not change the summed squared error $\Delta\mathbf{Tp}^{\mathrm{T}}\Delta\mathbf{Tp}$ of the pose. Hence the joint j would not have potential to compensate for the whole positional error of that particular pose.

Using equation 4.8 the measure of the error compensation potential of the j^{th} joint on $\mathbf{n}$ data samples has been implemented as

$$\mathbf{M}_{\mathrm{j}} := \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{m}_{\mathrm{j}}(\theta_{\mathrm{N}}^{\mathrm{i}}, \mathbf{P}_{\mathrm{C}}^{\mathrm{i}}) \tag{4.10}$$

which sums the absolute measures of each data sample for this particular joint. For all six joints of the PUMA 761 the measure of the compensation potential over n data samples can then be written as

$$\mathbf{M} := \sum_{\mathbf{i=1}}^{\mathbf{n}} \left| \mathbf{J}^{\mathrm{T}}(\mathbf{g}(\theta_{\mathrm{N}}^{\mathrm{i}}))\boldsymbol{\Delta}\mathbf{Tp}^{\mathrm{i}} \right| \tag{4.11}$$

with $\mathbf{M}$ being the vector that receives the measured values for each joint. Eventually, the joint that corresponds to the element of $\mathbf{M}$ with the largest value is selected for further evolutionary refinement of its correction model. (Note again, that the superscripted i in both previous equations is an index and does not represent exponentiation.)

4.5 Direct learning of joint correction models

An alternative approach to the evolution of joint correction models controlled by a distal teacher (the forward model) is the direct learning of the joint correction models. Direct learning means that not the performance index of the whole kinematic model as with the distal result of learning is subject to minimisation, but the direct error between nominal and calibrated (false target) joint configurations. Therefore in the remainder of this work the procedure of direct learning of correction models will also be referred to as direct joint error learning. Contrary to distal supervised learning the calibrated target joint configurations are explicitly given. The calibration data set containing nominal and calibrated joint configurations $(\theta_{\mathrm{N}}, \theta_{\mathrm{C}})$ is obtained using the nominal inverse model from the nominal and corrected poses $(\mathbf{P}_{\mathrm{N}}, \mathbf{P}_{\mathrm{C}})$

respectively[40]. An advantage of this approach as opposed to distal supervised learning is a computationally more efficient evaluation since the global performance index (equation 4.6) involving the evaluation of the whole DH model does not need to be computed. Instead the evolution of the $\mathbf{i}^{th}$ joint correction model is driven by its raw fitness measure

$$\mathbf{r}(\mathrm{i}) := \sum_{\mathbf{k}=\mathbf{1}}^{\mathbf{n}} \left| \theta_{\mathrm{C}}^{\mathrm{k}}[\mathrm{i}] - \mathbf{h}_{\mathrm{i}}(\theta_{\mathrm{N}}^{\mathrm{k}}[\mathrm{i}]) \right| \tag{4.12}$$

being the summed absolute error between the targeted calibrated joint configuration and the joint configuration computed by the correction function (equation 4.1) from the corresponding nominal joint variable for $\mathbf{n}$ calibration samples (Note that k in equation 4.12 is an index and no exponent). As this objective of minimisation is decoupled from the global performance index of the whole model, the joint correction models can be evolved independently for each joint. Since calibration data $(\theta_{\mathrm{N}}, \theta_{\mathrm{C}})$ from inverting the nominal and corrected poses is provided for all joints simultaneously, this property enables the parallel evolution of the joint correction models in GP systems being implemented on distributed computers.

Principally, the calibration set-up illustrated in Figure 4-5 applies to this calibration method too. However, the fitness evaluation within the populations needs to be implemented as measuring the raw fitness described in equation 4.12 and data preparation procedure involves the inverse transformation of the corrected poses $\mathbf{P}_{\mathrm{C}}$ into calibrated joint configurations θ_{C}.

4.6 Summary

This chapter presented details of a general genetic programming method for static inverse positional calibration of industrial robots. The basic idea of this approach is to evolve symbolic joint correction models so as to compensate for positional error of the end-effector of the robot when sent to offline generated poses. Two symbolic calibration methods have been proposed which basically differ in the format of the

[40] In this work only positional measurements were taken from the robot tool. The calibrated joint configuration of an end-effector pose (3 DOF) is hence computed from the nominal orientation and corrected position using the nominal inverse kinematic model. In case of full pose measurement (6 DOF) the calibrated joint configurations of a pose are obtained from the corrected orientation and corrected position.

calibration (or training) data and in the implementation of the fitness evaluation. The first method evolves (more specifically co-evolves) and evaluates the correction models in context of a nominal kinematic forward model being a distal teacher. The underlying concept of distal supervised learning being a general approach for learning inverse models is used to establish joint correction functions, which model the error of the nominal inverse kinematic model. Learning of these functions (more specifically the correction models) is performed sequentially by implementing a co-evolutionary scenario with the objective to minimise the performance index being a measure of the total positional error of the robot end-effector. In the second method the evolution of the correction models is performed by direct learning. Instead of evaluating the impact of an individual correction model on the total positional error of a kinematic model, the evolution is driven by the direct error between nominal and calibrated joint configurations. The implementation of a nominal inverse kinematic model is required to obtain the calibrated joint configurations from corrected training poses. The advantage of the direct learning method is a more efficient evaluation of the individual fitnesses, and a potential parallel (possibly distributed) implementation of the evolution of correction models for each joint.

Chapter 5

Implementation of the evolutionary calibration system

This chapter presents software implementation details of the new symbolic calibration concept developed in Chapter 4. Object oriented concepts have been applied to design the calibration system and the relationships between its components. First general issues for the design of the genetic programming system are discussed. Then the main object structures, which constitute the calibration system, are described.

5.1 GP implementation issues

An inherent property of evolutionary computation is multiple occurrence of genotypic material[41], which implies multiple and usually redundant evaluation. For standard genetic programming this means that several equal trees, which occur in different or within the same individuals, have to be traversed more than once. This is necessary for programs in which the result of the evaluation depends on the context. For example the control strategy of an artificial ant on the "Santa Fee Trial" is coded as a tree of motion commands [49]. The execution of those commands moves the ant

[41] Equal chromosome fragments may occur in individuals across the population. This is mainly due to the crossover operator. In genetic programming equal trees particularly small trees are often generated during population initialisation or by subtree mutation.

relatively to its current position in the toroidal grid. Thus evaluating the same control strategy in a different context results in different positions. However, in symbolic regression as used in this work the evaluation of mathematical expressions with the same arguments will always produce the same results. Hence a multiple evaluation of equal subtrees is very inefficient. In order to avoid this inefficiency equal subtrees could be shared as illustrated in Figure 5-1. The subtree is evaluated only once and the result reused for the evaluation of all trees sharing this subtree

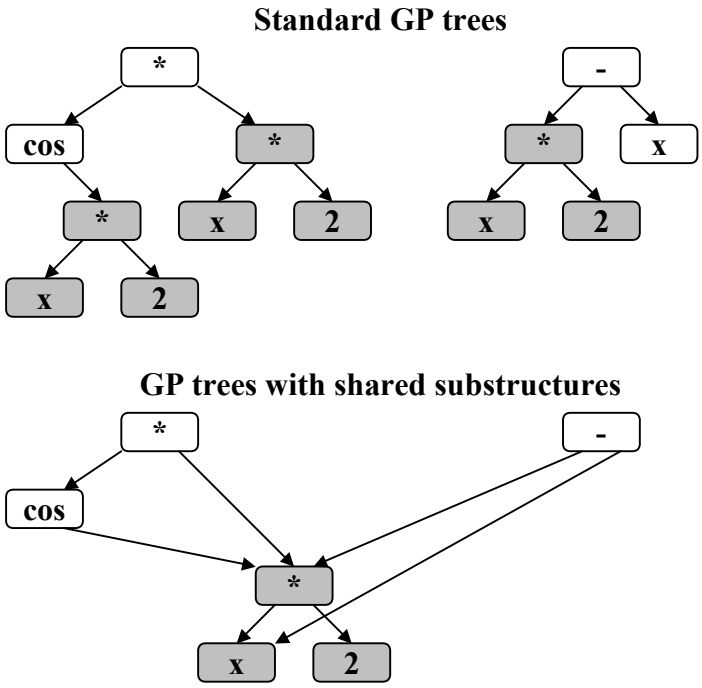


Figure 5-1: Ways of dealing with equal subtrees in genetic programming

In the GP implementation described in this chapter equal subtrees are shared between individuals across the population enabling them to be evaluated in parallel. The similarity between tree generation and tree evaluation in terms of beginning from the leaf nodes up to the root node (bottom-up tree generation, depth-first evaluation) has been exploited by storing the nodes in linear lists. The position of each node in the list corresponds to the order they were created with the first created node being on the bottom and the last node on the top of the list (Reverse Polish Notation). Thus nodes implement a dual representation as being tree nodes and list nodes as illustrated in Figure 5-2. Two lists are used to register the nodes of a population: one for terminals and for non-terminals i.e. functions (Note that these lists are not the terminal and non-terminal sets used for tree generation described in

section 3.3.2). Terminals are kept in a separate list since they do not change their value depending on other nodes. In order to evaluate a population it is necessary to specify particular values for the variable(s) in the terminal list and to evaluate each node in the non-terminal list sequentially bottom-up beginning from the first node created up to the last. During the evaluation each node stores its evaluated result, which can subsequently be accessed from other nodes that have been generated past this node. Thus the results of previous evaluations are reused which avoids multiple evaluations of shared subtrees.

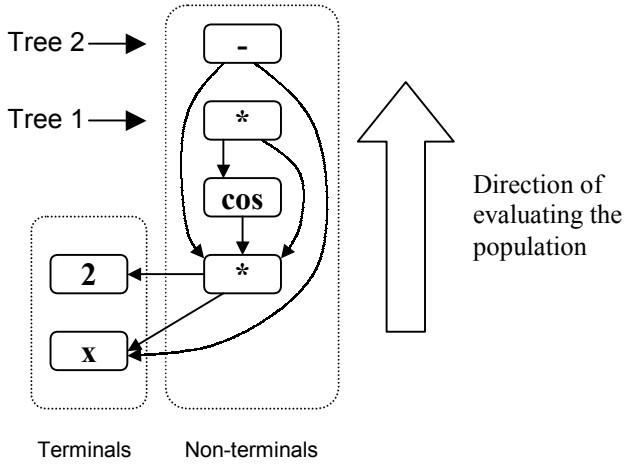


Figure 5-2: Internal dual representation of a population: to support efficient evaluation GP tree nodes are elements in a linear list arranged corresponding to the order of their creation (last created node on top)

The evaluation result of each tree in the population can be taken from the respective root node in the non-terminal list shown in Figure 5-2 (Note that the logical structure of the trees has not been changed). In this way the whole population is linearly evaluated avoiding inefficient multiple evaluation of common subtrees. Moreover, linear evaluation is computationally much more efficient than recursively traversing each individual tree in the population. However, compared to standard GP implementing individual tree evaluation the maintenance of shared substructures as explained above also involves a higher administrative overhead. A database of nodes currently used in all trees throughout the population must be established to enable all tree-producing mechanisms such as initial tree generation, mutation and crossover operator to find previously generated nodes. A garbage collection mechanism for example based on reference counting needs to be implemented to avoid unnecessary

evaluation of unused nodes. This mechanism keeps only those tree nodes in the database, which are either root of a tree in the population, or which are referenced by other trees.

The programming language chosen for this project was C++ [74] although LISP was initially considered. However, C++ was eventually selected since it exhibits better runtime performance and offers object oriented concepts, which permit a modular description of the very complex program and data structures used for the application of genetic programming in this work. In contrast to C++ the modelling and implementation of shared structures and hence the simultaneous evaluation of all individuals in a population is more complex using LISP.

5.2 Design and implementation of the main calibration system components

This section describes implementation details of the evolutionary calibration system developed in this work. This system is generally applicable to symbolically generate correction models to calibrate any manipulator. Due to the complexity of the whole software system only the main components (C++ classes) are listed and their structure briefly described. Also, only the main declarations (predominantly interfaces) relevant to explaining the functionality of the C++ classes are described.

Figure 5-3 depicts the dependencies and relationships of the C++- classes for the main components based on the illustration of the basic calibration procedure in Figure 4-5. This graph shows the inheritance, membership and relational information about the system classes used in the calibration system. It also shows the hierarchical structure of the calibration system, i.e. the decomposition of the main system into sub-components which are themselves aggregated from less complex components. References (implemented as pointers) between class instances (also known as objects) are used to send messages to the referenced object (by calling a corresponding method) to trigger a certain event (for instance from the calibration system object to a *gp_system* object to start the breeding process). Figure 5-3 also depicts the inheritance hierarchy (arrows point in direction of inheritance source) of system classes, if applicable. Furthermore the system classes are graphically grouped into different categories to illustrate their functionality and application context. The

evolutionary calibration system is mainly divided into classes, which implement the tree representation, the genetic programming mechanisms and the system classes controlling the calibration.

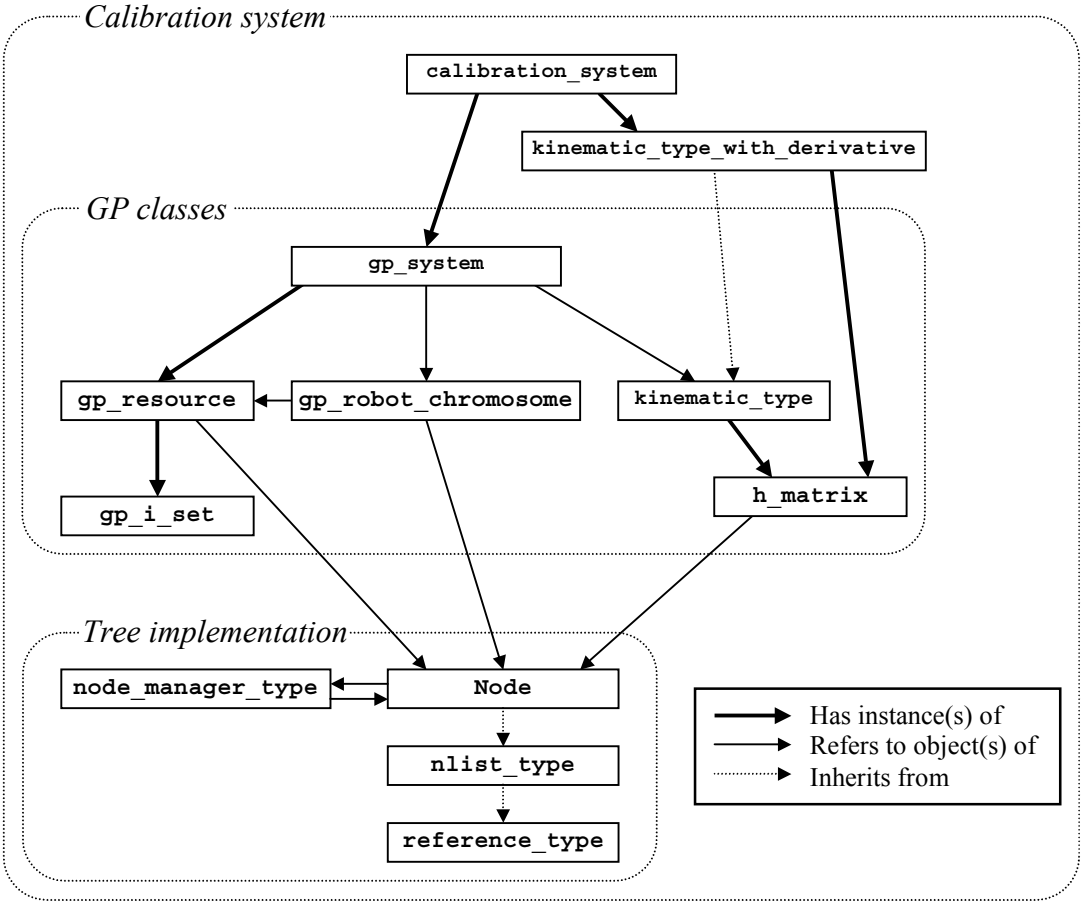


Figure 5-3: Combined dependency digraph of main C++ classes of the evolutionary calibration system

5.2.1 Tree implementation

The dual representation of GP tree / linear list shown in Figure 5-2 is implemented by four classes. As can be seen in Figure 5-3, the only interface class for the genetic programming algorithm to operate on this representation is the core class *Node* (see Table 5-1). This class implements a GP tree node including the evaluation algorithm described in section 5.1. The functionality's of the other three tree implementation classes are entirely transparent for the GP algorithm and are hence not described in detail. The class *nlist_type* inherited by *Node* implements an element of a double-chained linear list with the required list manipulation methods (insertion, deletion).

Hence each tree node is at the same time an element in a linear list. This implementation establishes the dual representation illustrated in Figure 5-2. The class *nlist_type* itself inherits the class *reference_type*, which provides a garbage collection mechanism based on reference counting as described in section 5.1. In order to establish shared substructures (in the case of equal subtrees) previously created equal nodes need to be found by the tree producing mechanisms (initial tree generating, mutation and crossover). This search is performed by the class *node_manager_type*, which implements a database to register nodes in two lists as described in section 5.1 and depicted in Figure 5-2. The node database and therefore the garbage collection is transparently operated by the class *Node*.

Class *Node*

Data structure	
`Constant`	Value of the node as string (only for variables, numbers and unary functions (SIN, COS etc.); otherwise the value is NULL.)
`Symbol`	Node descriptor. To identify the kind of the node: Possible values: PLUS, MINUS, MUL, PDIV, PLOG; and in conjunction with the value of `constant`: VARIABLE, NUMBER and FUNCTION.
`node_list`	Pointer to the node database
`Lnext`	Pointer to left subtree
`Rnext`	Pointer to right subtree

Methods	
`Node(name,sym,l,r, nl)`	Constructor: initialises data structure: `constant=name, symbol=sym, lnext=l, rnext=r, node_list= nl`, and registers the node in database `node_list`.
`~Node()`	Destructor: releases used resources and deletes the instance from the node database.
`get_node(name,sym, l,r)`	Retrieves a node. First the database `node_list` is searched for a node with a data structure corresponding to the parameter values. If no node is found it will be created. In that way the node database is operated entirely transparently. This is the only method used to generate tree nodes.
`output_infix( stream)`	Outputs the expression rooted at the current node into the file *stream*.
`calculate_list();`	Bottom up evaluation (as illustrated in Figure 5-2) of the entire expression stored in the node database.

Table 5-1: Structure of class *Node*

5.2.2 Calibration system structures

The actual genetic programming mechanisms are implemented in class *gp_system* (Table 5-7), which encapsulates an instance of the class *gp_resource* (see Figure 5-3). The class *gp_resource* (Table 5-2) provides the terminal and function set (of type *gp_i_set*) as well as the implementation of basic tree generation (GROW, FULL) and validation (maximum tree depth etc.) methods used by the genetic programming algorithm.

Class *gp_resource*

Data structure	
`node_db`	Pointer to node database of the population
`Terminals`	Set of terminals
`Functions`	Set of functions

Methods	
`gp_resource()`	Constructor, creates empty node database and initialises terminal and function set.
`create_tree(d, fulldepth)`	Creates trees of depth *d*; the *fulldepth* parameter determines the tree generation method: *fulldepth*==false: GROW method *fulldepth*==true: FULL method
`valid_tree(node)`	Checks if *node* fulfils tree generation constraints (minimal and maximal tree depth).
`tree_depth(node)`	Returns the depth of the tree rooted at *node*.

Table 5-2: Structure of class *gp_resource*

As shown in Figure 5-3 the class *gp_system* also refers to an instance of class *kinematic_type* (Table 5-5), which implements a general forward kinematic model (geometric) required for evaluating correction models evolved using distal supervised learning. The kinematic model equations are represented as a homogenous matrix of symbolic expressions (GP trees), which is implemented in class *h_matrix* (Table 5-3). For this representation of the model equations the class *Node* has been reused due to its efficient evaluation mechanism (*calculate_list()* method in Table 5-1). The model equations have been established from Denavit-Hartenberg (DH) specifications stored in string matrices (see Table 5-4). The elements of such a string matrix are strings representing the corresponding terms of a DH matrix. In order to build a kinematic model DH of a particular robot the string matrices for all links are parsed[42] into matrices of symbolic expressions, which are symbolically multiplied according to their order. By symbolic multiplication of two trees is meant the creation of a new node representing multiplication with the two factor trees being child (or argument) trees. Symbolic addition is performed accordingly. Both, symbolic multiplication and addition are then used to implement the matrix multiplication operation in method *r_multiplication(...)* (Table 5-3).

Class *h_matrix*

Data structure	
`matrix`	3×4 matrix of *Node* pointers, represents the kinematic equations in a homogenous matrix.
`node_set`	Pointer to the node set from which the equations in *matrix* are constructed.

Methods	
`h_matrix(strm,set)`	Constructor: Creates matrix of symbolic expressions by parsing the string matrix *strm* (see Table 5-4 for example) into member *matrix*. The variable *node_set* is initialised with *set*.
`~h_matrix()`	Destructor: Deletes the symbolic expressions in *matrix*.
`r_multiplication(strm)`	Parses the string matrix *strm* (example matrix Table 5-4) and right-multiplies the result symbolically with the current expressions in *matrix* and stores the product in *matrix*. This method is used to multiply the single DH matrices when building the overall kinematic robot model.
`add_tool(t)`	Adds the translation *t* of the tool tip relative to the tool frame.

Table 5-3: Structure of class *h_matrix*

[42] Expression parsing has been implemented using recursive descent. See [3] for a description of parsing principles.

```
String_matrix_type dh1=
{{"cos(theta1)","-cos(PI/2)*sin(theta1)","sin(PI/2)*sin(theta1)" , "0"},
 {"sin(theta1)","cos(PI/2)*cos(theta1)" ,"-sin(PI/2)*cos(theta1)", "0"},
 {"0"          ,"sin(PI/2)"             ,"cos(PI/2)"             , "0"}
}
```

Table 5-4: An example of a string matrix: DH matrix for the first link of the PUMA 761

An instance of class *kinematic_type* is initialised with a sequence (array) of DH matrices represented as string matrices (see Table 5-4). All string matrices (See also source code at page 161 in the appendix) are parsed into matrices of GP trees and, corresponding to their order, subsequently multiplied together (calling method *r_multiplication(...)* of member *matrix* for each string matrix) to constitute the overall kinematic model (stored in *matrix*) of the particular robot. In this way the kinematic model of any manipulator can easily be established from its DH description.

Class *kinematic_type*

Data structure	
`matrix`	Matrix containing the kinematic equations as symbolic expressions.
`joints`	List of direct accessible nodes of the joint variables (needed for assigning the joint values when evaluating the kinematic model).

Methods	
`kinematic_type(node_db,string_matrices[],n,tool);`	Constructor: Initialises *matrix* with the kinematic model from DH description stored in field *string_matrices[]* of length *n* using node database *node_db*. The *tool* parameter determines the dimension of the tool used for calibration. The member variable *joints* is initialised with the nodes representing the joint variables.
`~kinematic_type()`	Destructor: Destroys the joint variable list *joints* and the trees stored in *matrix*.
`compute_forward_kinematic(dataset[],n)`	For each of the *n* joint configurations in *dataset[]* the corresponding end-effector position vector is computed and assigned to the particular dataset sample.
`compute_position_error(dataset[],n)`	Returns the performance index (equation 4.6 on page 63) of *n* samples of calibration data in *dataset*[] (joint configurations and corresponding target position).

Table 5-5: Structure of class *kinematic_type*

The genetic programming algorithm implemented by class *gp_system* (Table 5-7) operates on a population of instances of the class *gp_robot_chromosome* (Table 5-6)

which encapsulates a single joint correction model as a tree constructed from instances of the class *Node*. Furthermore this class implements mutation and crossover operators, which are applied to the correction model when undergoing genetic modification.

Class *gp_robot_chromosome*

Data structure	
`theta_index`	Indicates the number of the joint the correction model is applied to. Since each instance of this class implements the correction model of one particular joint, this information is required for the evaluation mechanism.
`data[]`	Set of calibration samples (joint configurations and target end-effector positions (see section 4.3).
`resource`	Pointer to *gp_resource* instance that defines the terminal and function sets linked to the population this particular chromosome belongs to. This information is particularly needed for mutation.
`joint`	Root of the GP tree representing the correction model.

Methods	
`gp_robot_chromosome(resource,max_depth,full_depth,j_index)`	Constructor: Links the instance to its joint *j_index* and to its terminal and function set in *resource* (type *gp_resource*) and prompts *resource* to create a tree using the parameters max_depth and *full_depth* (see Table 5-2).
`~gp_robot_chromosome()`	Destructor: Destroys the GP tree rooted at the node stored in *joint*.
`mutation()`	Performs mutation on the tree rooted at the node stored in *joint* according to globally specified parameters (see Table 5-8).
`crossover(c)`	Performs crossover with chromosome *c*.
`apply_corrections(dataset[],n)`	Core method: Applies correction to each of the *n* data samples in *dataset*[]. Corrected in each sample will be the value of the joint specified by the member variable *theta_index*. This method is used after calibration to correct offline generated poses.
`equals(c)`	Returns true if this instance is equal to chromosome *c*. Otherwise it returns false. This method is used during the initialisation of the population to prevent the emergence of equal chromosomes.
`evaluate(n,km)`	Returns the fitness value, which is the performance index of *n* samples of the calibration data stored in member variable *data[]* using the kinematic model *km* (type *kinematic_type*).
`evaluate_population(p[],pn,ds[],n,km);`	Parallel evaluation of all *pn* chromosomes in population *p[]* using the kinematic model *km* on *n* calibration data samples stored in *ds[]*.

Table 5-6: Structure of class *gp_robot_chromosome*

The class *gp_system* (Table 5-7) encapsulates and implements the basic genetic programming algorithm operating on a population of correction models. This includes control over population administration (i.e. initialisation, evaluation) and the actual breeding mechanism based on tournament selection. Each instance of this class within the calibration system (implemented by class *calibration_system* shown in Table 5-10) evolves a correction model for one particular joint.

Class *gp_system*

Data structure	
`gp_set`	Function and terminal set linked to this particular genetic programming system.
`kinematic_model`	List of direct accessible nodes of the joint variables (needed for assigning the joint values when evaluating the kinematic model).
`population`	Field of chromosomes of type *robot_gp_chromosome* representing the population.
`new_population`	Population of the offspring chromosomes.
`theta_index`	Joint index linked to this GP system.
`kinematic_model`	Kinematic model of type *kinematic_type* required for evaluating the correction models (in distal supervised learning).

Methods	
`gp_system(ds[],n,km,ti);`	Constructor: Links the GP system to the joint denoted by *ti*, initialises the population with chromosomes, *n* calibration data samples in *ds[]*, and a reference to the kinematic model (of class *kinematic_type*).
`~gp_system()`	Destructor: Destroys the population of correction models.
`init_half_and_half()`	Initialises the population using the RAMPED HALF & HALF method (section 3.3.3).
`already_in_population(ch,n);`	Scans the first *n* chromosomes in member *population* for occurrences of chromosome *ch* and returns true if one was found. Otherwise the function returns false. This method is used for the initialisation of the population to prevent multiple occurrences of chromosomes.
`breed_until_improvement(ds[],n,f,g,max_gen)`	Breeds new populations until the performance index on *n* calibration data samples in *ds[]* could be improved (see also Figure 4-6) or the current generation *g* exceeds the maximum number *max_gen* of generations. Status information (fittest individual, current generation) will be logged into file *f*.
`correct(ds[],n)`	Calls the method *apply_correction(ds[],n)* (see Table 5-6) from the best correction model in the population.
`write_statistic(f)`	Stores the best performing correction model in file *f*.

Table 5-7: Structure of class *gp_system*

The run of the genetic programming algorithms in the individual instances of class *gp_system* is globally controlled by the parameters listed in Table 5-8:

GP Parameter	Description
POPULATION_SIZE	Number of chromosomes in a population
NUMBER_GENERATIONS	Number of populations initially generated. This is a parameter, that can be increased interactivly by the user in order to continue the evolution.
TOURNAMENT_SIZE	Number of chromosomes competing in the tournament .
INITIAL_MAX_TREE_DEPTH1	Initialisation of a population is performed using the RAMPED HALF&HALF method. This parameter describes the maximal tree depth at the beginning of the ramp.
INITIAL_MAX_TREE_DEPTH2	Maximal tree depth at the end of the ramp
MAX_TREE_DEPTH	All tree producing operators (initial tree generations, crossover, mutation) are constrained not to generate a tree with a larger depth than described with this parameter.
MIN_TREE_DEPTH	The tree generation of the GROW method and mutation operator is constraint to produce trees of a minimum depth in order to prevent the populations from occurrences of short trees or terminals which may not have the potential to sufficiently model the joint error.
INVALID_ATTEMPTS	This parameter controls the number of attempts to create a valid tree by crossover and mutation. If this number is exceeded the respective parent(s) will be reproduced for the next generation.
CROSSOVER_RATE	Probability of applying crossover.
MUTATION_RATE	Probability of applying mutation.
TERMINAL_CROSSOVER_RATE	This parameter determines the probability of terminals being crossover points (A small value characterises a preference for subtree crossover which has the potential to introduce larger changes to the offspring).
TERMINAL_MUTATION_RATE	Probability of mutating terminal nodes.
SHRINK_MUTATION_RATE	The probability of mutating trees by replacing selected subtrees with randomly generated trees of lower depth.

Table 5-8 : GP parameters used by the calibration system

The overall calibration system is implemented by class *calibration_system* (Table 5-10), which encapsulates 6 autonomous instances of class *gp_system* (one for each joint of the PUMA 761 robot (see also Figure 4-5)) and an instance of class *kinematic_type_with_derivative* (Table 5-9). The class *calibration_system* also implements the joint selection mechanism used by the distal supervised learning

method described in section 4.4.2. The required measure of the compensation potential of each joint (equation 4.11) is implemented by the class *kinematic_type_with_derivative.*

Class *kinematic_type_with_derivative*

Data structure	
`der1,der2,der3, der4,der5,der6`	Homogenous matrices of type *h_matrix* (node matrix) representing the derivatives of the kinematic model matrix with respect to joint 1-6.

Methods	
`kinematic_type_with_derivative(node_db,string_matrices[],n,tool)`	Constructor: Parameters are passed on to the constructor of the inherited class *kinematic_type* (same parameter list) to initialise the kinematic model used throughout the calibration system. Initialises the matrices *der1* - *der6* with the derivatives the kinematic equations according equation 5.1. These matrices are established by reusing the symbolic multiplication mechanism inherited from class *kinematic_type*. The parameter *tool* expresses the translation of the tool end point to the origin of the tool frame.
`get_joint_with_most_p(ds[],n,f);`	Returns the number of the joint with the most potential for error compensation (described in section 4.4.2) based on *n* data samples in *ds[]*. A data sample in *ds[]* consist of the currently evolved joint and corrected end-effector position (target). The result of the computation is logged into file *f*.

Table 5-9: Structure of class *kinematic_type_with_derivative*

The necessary derivatives of the kinematic model equations with respect to each joint are obtained from

$$\mathbf{D}_j = \left(\prod_{i=1}^{6} \mathbf{K}_{ij} \right) \mathbf{P} \qquad \text{with } \mathbf{K}_{ij} = \begin{cases} \frac{\partial \mathbf{A}_i}{\partial \theta_j} & ;\ i{=}j \\ \mathbf{A}_i & ;\ \text{else} \end{cases} \tag{5.1}$$

where $\mathbf{D}_j$ is the matrix derivation of the homogenous overall kinematic model matrix with respect the j^{th} joint, where $\mathbf{P}$ is the 4×4 tool transformation matrix and $\mathbf{A}_i$ the DH matrix of the i^{th} link (see also appendix section A.2). Computation of the model derivatives according to equation 5.1 is performed symbolically by class *kinematic_type_with_derivative* (Table 5-9). That is the mechanisms inherited (see class graph in Figure 5-3) from class *kinematic_type* (Table 5-5) for constructing a kinematic model from an array of string matrices have been reused to initialise the member matrices `der1-der6` (Table 5-9). For each joint j the required matrix

derivative $\frac{\partial \mathbf{A}_i}{\partial \theta_j} = \begin{bmatrix} \frac{\partial \mathbf{R}_i}{\partial \theta_j} & \frac{\partial \mathbf{X}_i}{\partial \theta_j} \\ \mathbf{0}^T & \mathbf{0} \end{bmatrix}$ ($\mathbf{R}_i$ is the 3×3 rotation matrix and $\mathbf{X}_i$ the 3×1 position vector of the i[th] link) in equation 5.1 is provided by a string matrix (see Table 5-4). The elements of such a string matrix are the derivatives (with respect to θ_j) of the elements of the i[th] DH matrix represented as strings. The model derivative $\mathbf{D}_j$ is then generated according to equation 5.1 using the model generation mechanisms (equation parsing and symbolic matrix multiplication) inherited from class *kinematic_type*. Again, this design enables an easy adaptation of the calibration system to robots with different kinematics, as only the kinematic parameters in the string matrices (see Table 5-4 for example) need to be textually edited prior to calibration.

Class *calibration_system*

Data structure	
`gp1,gp2,gp3,gp4, gp5,gp6`	6 instances of class *gp_system*, one for each joint.
`der`	Instance of class *kinematic_type_with_derivative*.

Methods	
`Calibration_system (ds[],n,km)`	Constructor: Initialises the six genetic programming systems for each joint. Parameters *ds[], n* und *km* are passed on to the constructor method of *gp_system* for each instance (see Table 5-7).
`ga(ds[],n,f)`	Main method: calling this method starts the calibration procedure. Parameters are the *n* calibration data samples in *ds[]* and the descriptor for the logfile *f*, which receives information about the calibration process. The implementation essentially corresponds to the algorithm in Figure 4-6.
`correct(ds[],n)`	When calibration has been performed, correction is applied by calling this method for *n* joint configurations in data set *ds[]*. This method subsequently calls the *correct()* method of each of the six GP systems to correct the respective joint value in each of the *n* data samples (provided a correction model has been evolved).
`write_statistic(f)`	Calls the *write_statistic()* of all 6 GP system instances.

Table 5-10: Structure of class *calibration_system*

Using the C++ class definitions described in this section the entire calibration process is performed by the compact C++ procedure shown in Table 5-11.

First, a kinematic model object (type *kinematic_type*) is created, which will be used for the preparation of calibration data and the evaluation of correction models

throughout the calibration using distal supervised learning. The parameter *PUMA_parametric* contains an array of string matrices (see Table 5-4), which describe the *6* DH matrices of the PUMA 761 industrial robot used for the calibration experiments in this work. The parameter *tool* describes the dimension of the tool used as being the translation of the tool endpoint from the origin of the tool frame.

Having initialised the kinematic model the calibration data set is prepared using the algorithm shown in Figure 4-3 and a file receiving information about calibration process (e.g. joint selection status and performance index) and results is opened. Subsequently, the main calibration system (*c_system*) is instantiated.

```
void main_gp(Node node_db)
{
   kinematic_type kinematic_model(node_db,PUMA_parametric,6,tool);
   dataset_type data[100];
   int data_samples=prepare_calibration_data(data,&kinematic_model);
   if (data_samples==-1) throw int(0);
                        //if data preparation was in error throw exception
   file_manager logfile("gp_logfile.txt","w+");
   calibration_system c_system(data,data_samples,kinematic_model);
   c_system.ga(data,data_samples,logfile.file);
   alter_joint_angles("t1.v2","updated_t1.v2",&c_system);
   alter_joint_angles("t2.v2","updated_t2.v2",&c_system);
   c_system.write_statistic(logfile.file);
}
```

Table 5-11: C++ implementation of the calibration procedure

The calibration is started by calling the *ga()* method of the calibration system (*c_system*) with the calibration data set, the number of data samples in the data set and the log file descriptor being the arguments. When the calibration has succeeded the evolved joint correction models within the GP systems in *c_system* are used to alter joint configurations stored in offline generated robot program files. For this the procedure *alter_joint_angles* is called for two VAL II files (*t1.v2* and *t2.v2* in the example) to apply corrections to the joint configurations stored in these files. The corrected joint configurations are then stored in the files *updated_t1.v2* and *updated_t2.v2* respectively, which are eventually uploaded into the robot controller. Finally, the results of the calibration procedure namely the evolved correction models are stored in the file *logfile*.

5.3 Summary

This chapter described software design and implementation issues of the evolutionary calibration method presented in Chapter 4. An evolutionary calibration software system has been developed, which is generally applicable to symbolically generate joint corrections to calibrate any manipulator. The structure of this calibration system and its main components has been outlined in this chapter. The representation of symbolic expressions is based on shared substructures implemented to improve the runtime performance of the calibration system. The application of this software to the positional calibration of a PUMA 761 manipulator is described in Chapter 6.

Chapter 6

Results from calibration experiments on a PUMA 761 manipulator

This chapter presents the experimental results of the evolutionary calibration method described in Chapter 4. The experiments were carried out on a 6 DOF Unimation PUMA 761 industrial robot. First the general context is explained i.e. the data measurement procedure, GP system set-up etc. Symbolic calibration is then performed on a set of measured data using both variants of the genetic programming method and the resulting joint correction models are presented.

6.1 Calibration set-up

First a calibration area within the workspace of the robot has to be established. In the experiments described in this chapter a calibration area of the form of a cuboid has been defined the edges of which are parallel to the axes of robot base frame. The dimension of this cuboid was defined to be $(x, y, z) =$ (844 mm, 582 mm, 1151 mm). The location of its corner, which is the closest to the origin of the robot base frame, has been defined in robot base frame co-ordinates as $(x, y, z) =$ (432 mm, 490 mm, -675 mm). The actual robot tool used for the experiments is displayed in Figure 6-1 and has been accurately measured using a 3 co-ordinate measurement machine. The position of the tool endpoint is described relative to the tool frame as the translation $(\Delta x, \Delta y, \Delta z) =$ (150.25 mm, 1.63 mm, 55.69 mm). As generally

stated for model-based calibration, all kinematic parameters may be identified using only position measurements if the measured points are not located along the tool axis [25]. This requirement is met by the tool shown in Figure 6-1 and also applies to the symbolic calibration method described in this work if correction models are to be generated for all joints. If a tool was used for which the endpoints were located along the local tool axis only $(x = 0, y = 0)$ no error could have been attributed to the last joint. In that case the partial derivatives of all components in the position vector with respect to joint 6 (used in the error Jacobian in equation 4.7) were zero resulting in a zero measure for all calibration poses. Hence no correction model would be generated for this joint.

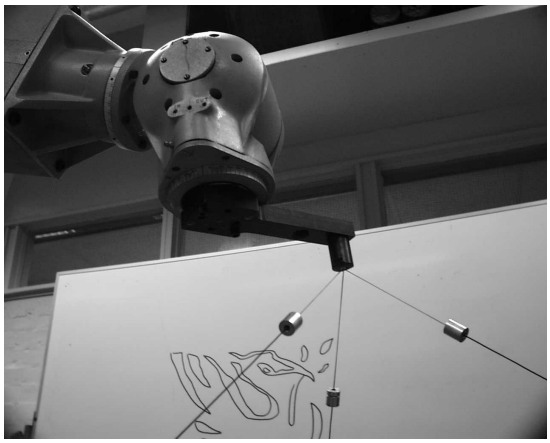


Figure 6-1: Robot tool used for experiments

6.1.1 Calibration data

For the calibration experiment a data sample set consisting of thirty 30 random poses within that the calibration area defined in section 6.1 was generated. In order to guarantee observability of the error in each joint it was made sure that all joints were involved when moving through these poses[43]. At a speed of 16% (of full speed, see equipment manual [75]) the robot was sent to these calibration poses and measurements of the actual positions of the tool endpoint were taken using the

[43] However, the full possible range of each joint was not considered since the calibration was performed local to the defined calibration area and restrictions in the tool orientation were imposed by the Robotrak measurement system.

Robotrak measurement device (see section A.1). The corresponding joint configurations of these poses were obtained directly from the VAL II system during the measurement procedure. Therefore there is no need to implement a nominal inverse kinematic function to convert the poses into joint configurations required for calibration. Each time the robot settles for a pose the corresponding joint configuration can be obtained by issuing the VAL II command `HERE #p` which stores the current pose as a precision point (6 joint angles) in variable `p`. The small program in Figure 6-2 was used to drive the robot to the calibration poses and to obtain the joint angles, while Robotrak took the positional measurements. Storing this program after its execution into a file automatically attaches all used variables and therewith the array `#cp[]` of generated joint configurations [72].

```
.PROGRAM t1
  FOR l = 1 TO 30
    MOVE mcp[l]
    DELAY 3
    HERE #cp[l]
  END
.END
```

Figure 6-2: VAL II program used to obtain measurements and joint configurations

Following the measurements of the calibration poses the three local frame points needed for the data transformation (see section A.1.1) were manually taught and subsequently measured by Robotrak.

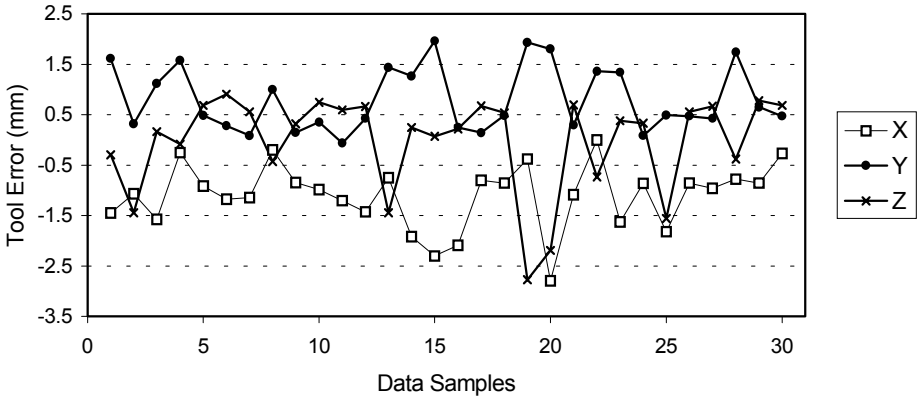


Figure 6-3: Positional error of the robot tool end point in X, Y and Z on the calibration data set prior to calibration

The positional error of the robot tool end point in X, Y and Z direction at the 30 calibration poses prior to calibration is shown in Figure 6-3. The absolute tool error (or tool deviation) being $\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$ at these poses is shown in Figure 6-4.

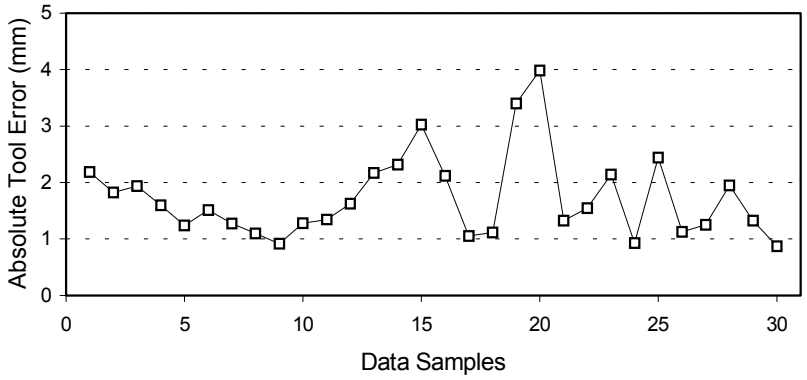


Figure 6-4: Absolute positional error of robot tool end point on the calibration data set prior to calibration

6.1.2 Validation data

In order to evaluate the generalisation capabilities of the evolved joint correction models another set of 30 random end-effector poses has been generated within the calibration area defined in section 6.1. The measured positional error of the tool endpoint at these poses prior to calibration is illustrated in Figure 6-5.

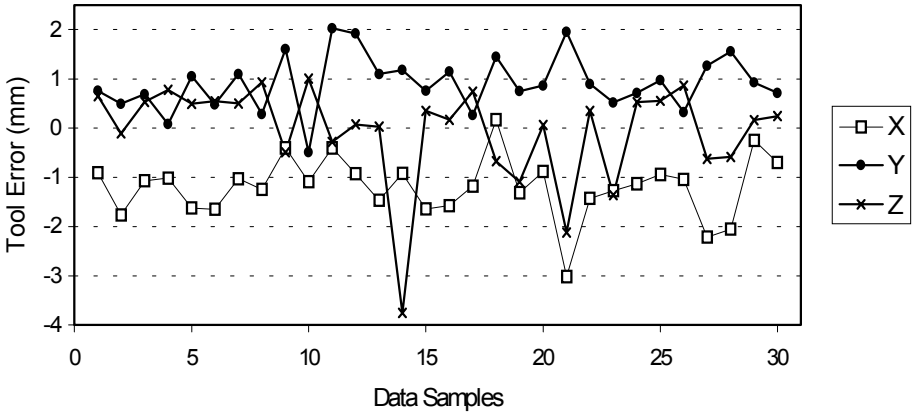


Figure 6-5: Positional error of the robot tool end point in X, Y and Z on the validation data set prior to calibration

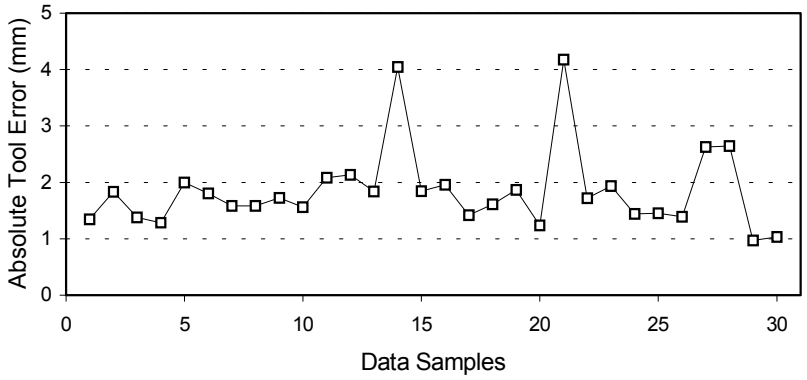


Figure 6-6: Absolute positional error of robot tool end point on the validation data set prior to calibration

6.2 Experimental symbolic calibration using Distal Supervised Learning

In this section the results are presented obtained from a symbolic calibration experiment using the distal supervised learning approach described in section 4.3. The GP system instances within the evolutionary calibration system (see Figure 4-5 and Figure 5-3) implement the generational evolutionary algorithm and tournament selection with elitism. The genetic programming parameters, which have been used in the experiments, are shown in Table 6-1 (see also Table 5-8 for description).

GP Parameter	Value
POPULATION_SIZE	300
NUMBER_GENERATIONS	1000
TOURNAMENT_SIZE	5
INITIAL_MAX_TREE_DEPTH1	3
INITIAL_MAX_TREE_DEPTH2	5
MAX_TREE_DEPTH	9
MIN_TREE_DEPTH	3
INVALID_ATTEMPTS	21
CROSSOVER_RATE	0.8
MUTATION_RATE	0.2
TERMINAL_CROSSOVER_RATE	0.2
TERMINAL_MUTATION_RATE	0.3
SHRINK_MUTATION_RATE	0.2

Table 6-1: GP parameters used in the calibration experiment using distal supervised learning

The terminal set $\mathbf{T}$ and the non-terminal set $\mathbf{F}$ were defined as:

$$\mathbf{T} = \{\theta, \Re\}$$

$$\mathbf{F} = \{+, -, *, \%, \mathrm{SIN}, \mathrm{COS}, \mathrm{SIGN}, \mathrm{SQRTp}\}$$

with θ being the joint variable, $\Re \in [\mathbf{0}, \mathbf{1}]$ a random ephemeral constant, with addition, subtraction, multiplication, the protected division $\%$, sine, cosine, the sign function and the protected square root (see section 3.3.1).

6.2.1 The calibration process

The process of evolving joint correction models was limited to 1000 generations (see other parameters in Table 6-1), the computation of which took about 15 minutes on a PC with an Intel Celeron™ processor running at 500Mhz (The time required to complete a calibration varies between different trials due to different structural complexity of the evolved expressions). The reduction of the performance index (equation 4.6) as being the fitness measure during the entire evolutionary process is shown in Figure 6-7. This figure also illustrates the discontinuity of the evolutionary progress. By evolving more accurate correction models the algorithm rapidly reduces the performance index within the first 50 generations followed by a long phase with minor improvements, and from generation 815 with a further significant reduction.

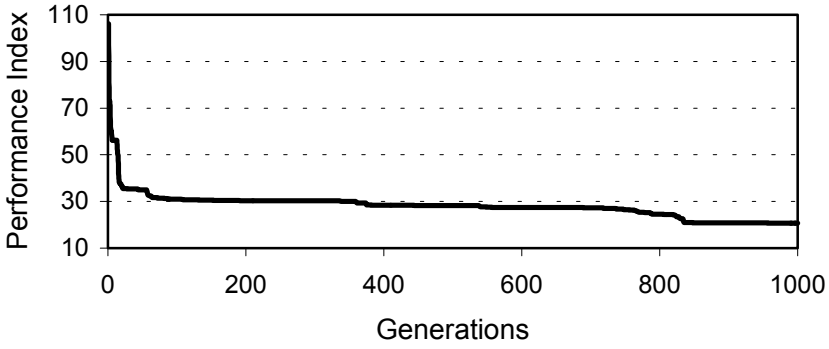


Figure 6-7: Performance index of the kinematic model during the evolution of the joint correction models

The individual components of the performance index during the entire evolutionary process are shown in Figure 6-8 documenting the contribution of the summed squared error in X, Y and Z direction over all data samples. The number of

a joint for which a correction model is being evolved at a particular generation is shown in Figure 6-9. The selection of a currently evolved joint correction model during the evolution is based on the current error correction potential of the joints shown in Figure 6-10 and Figure 6-11. It can be seen from these figures that rapid improvements of the performance index occur particularly after the algorithm switched to another joint to proceed refining its correction model.

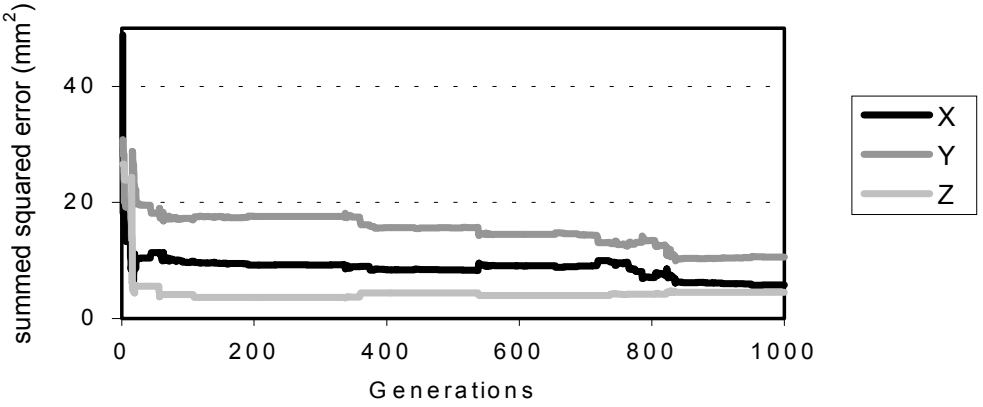


Figure 6-8: Components of the performance index (summed squared error in X, Y and Z between target pose and evolved pose over all 30 data samples) during the evolution of the joint correction models

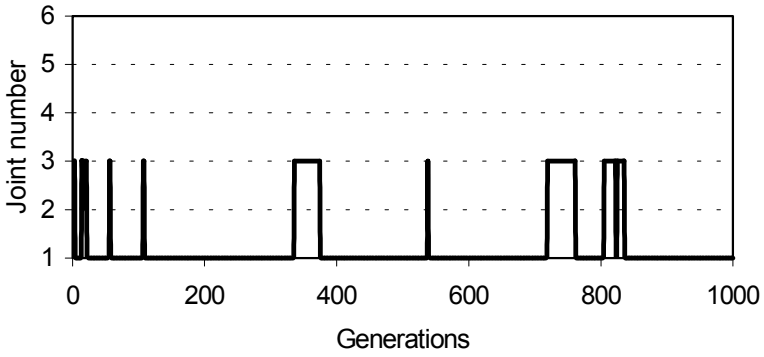


Figure 6-9: Joint selection performed by the calibration system during the evolution

Figure 6-9 shows the behaviour of the joint selection mechanism (see also section 4.4.2) used by the calibration system and the generations spent to evolve correction models for particular joints. The selection of a particular joint, for which a correction

model is to be evolved, is based on the current error correction potential (see equation 4.11) shown in Figure 6-10 and Figure 6-11. As it can be seen from these figures, only the joints 1-3 have been selected during the evolution. This suggests that based on the particular calibration poses used in this experiment and by neglecting the small changes in tool orientation, the positional error of the robot tool end point could be reduced by evolving correction models for joint 1-3 only.

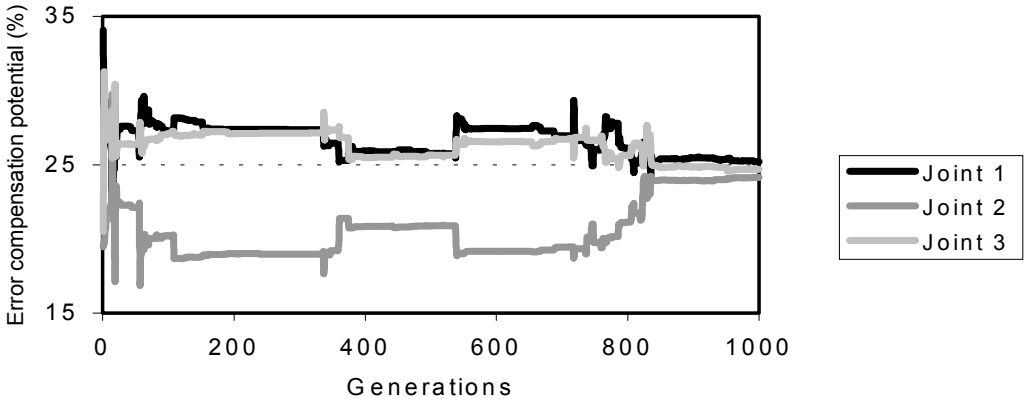


Figure 6-10: Error correction potential of joint 1-3 based on equation 4.11 during the evolution of correction models

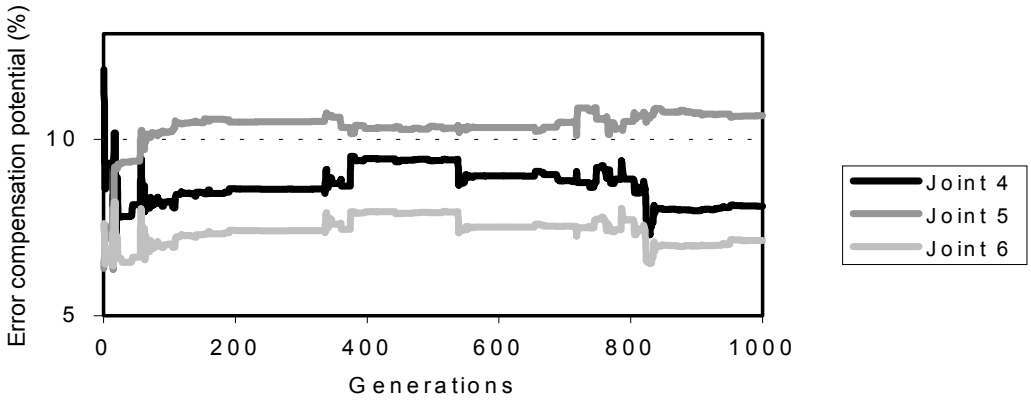


Figure 6-11: Error correction potential of joint 4-6 based on equation 4.11 during the evolution of correction models

6.2.2 Calibration results

In this section the results of correcting the positional error of the robot tool endpoint by the evolved joint correction models are documented. Figure 6-12 graphically depicts the absolute error of the tool endpoint on the calibration data set prior to calibration and using the correction models for joint 1-3 (see Table 6-2).

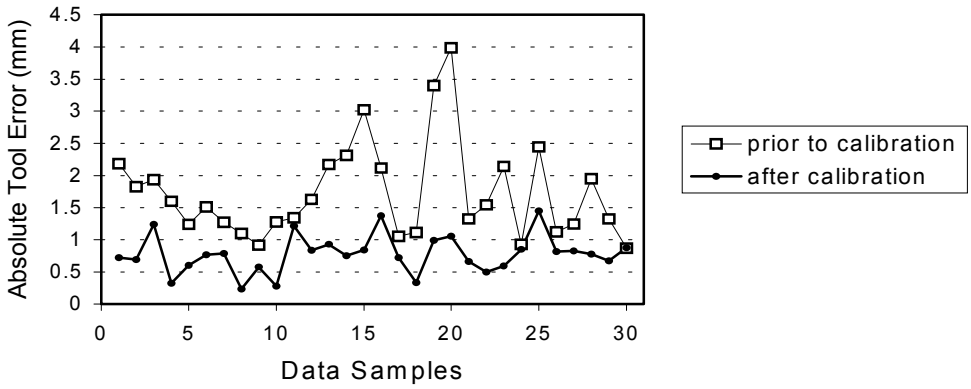


Figure 6-12: Comparison of the absolute positional error of the robot tool end point on the calibration data set prior and after calibration

The error reduction property of the evolved correction models on poses not included in the calibration is illustrated in Figure 6-13. This figure shows the reduction of the absolute positional error of the poses from the validation data set in comparison to the uncalibrated manipulator.

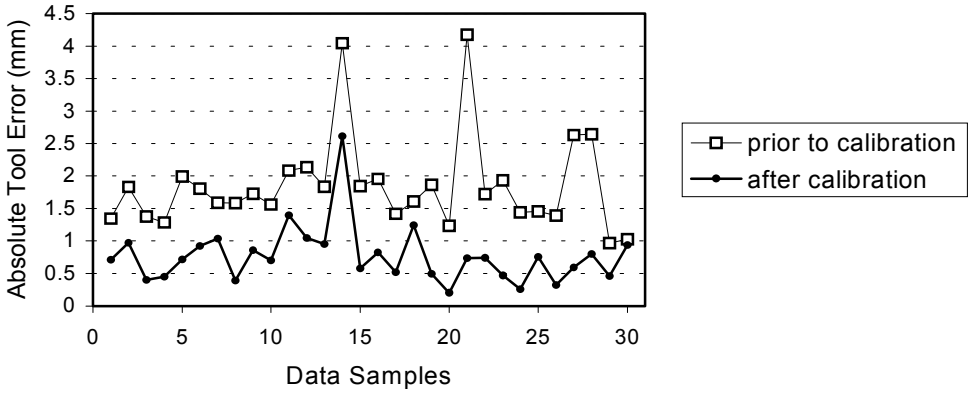


Figure 6-13: Comparison of the absolute positional error of the robot tool end point on the validation data set prior and after calibration

Joint	Evolved symbolic expression
1	(SIGN(COS((0.0588503524887844 5%theta*COS(0.0585776238288522))-theta%COS (SIN(0.0585776238288522))))*COS(COS((COS(theta) %theta*theta%theta)-theta% COS(0.278844630268257 7*theta)))*(COS (theta)*COS(SIN(0.05852710043641468))*0.9934800561540573-COS (theta))%(COS(COS(COS(theta)-0.2787233039338359 *theta-theta% (0.07754042176580096%0.6930728629413739))) %COS(COS((COS (0.9036369823297831)%theta*COS(0.9041817529831843)%theta)))))
2	SIN((COS(theta)*(0.0004272591326639607%theta)%0.5609912411877804))
3	((((0.9522924588763085-0.7845088045899838%0.8250679036835841) *COS(SIN ((SIGN(0.8828394421216468)%SIN(theta)*0.1264239631336405))))%(((SIN(theta)*th eta*0.1264239631336405)%(0.391308328501236%SIN(theta))+theta+(0.3767509994 811853*0.3541520584734641)%(0.1264239631336405%SIN(theta)-(theta-0.271553697317423)))*0.1264239631336405))%(SIN(0.6908403881954406)*(SIN(SI N(0.5986205633716849)%SIN(theta)-theta)+SIN(SIN(0.6535240180669576))+ SIN(theta)+theta+SIN(theta)%0.3541520584734641))

Table 6-2: Evolved symbolic expressions of joint correction models for joint 1-3 established using distal supervised learning

In Table 6-2 the joint correction models represented as symbolic expressions generated by the calibration system using distal supervised learning are listed. The results from the calibration procedure (the performance index reduction of the calibration model during the evolution) and the actual positional measurements taken from the robot tool prior and after calibration are summarised in Table 6-3.

	Prior to calibration	After calibration
Performance index (on calibration data)	106.203676	21.185473
Mean positional tool end point error on calibration data set (mm)	1.730961	0.776642
Mean positional tool end point error on validation data set (mm)	1.850646	0.770033

Table 6-3: Calibration results using the correction models (Table 6-2) evolved by distal supervised learning

Figure 6-14 shows the graphs of the evolved correction models Table 6-2 across the calibration range (limited by the calibration area) of their respective joints. It illustrates the quality of the symbolic expressions in modelling the targeted correction values for the sets of calibration and validation data. The reader may recall that these target values of the calibration data set were not explicitly provided in this method. Instead the distal performance of the kinematic model including all three correction models has driven the evolution (see section 4.3). Particularly interesting is the graph of the correction model evolved for joint three. The joint values

computed by VAL II from the nominal (used for the calibration) and corrected end-effector positions are expressed as positive numbers for quadrant 1-2 and as negative numbers 3-4. As the joint values of the calibration data set are located on both sides of the transition point from the second to the third quadrant, the calibration was performed for two separate intervals $[-\pi \ldots \mathbf{a}]$ and $[\mathbf{b}\ldots\pi]$. In the interval $[\mathbf{a}\ldots\mathbf{b}]$, which was not included in the calibration, the correction model evolved in this particular experiment shows undesired behaviour by starting to oscillate and to produce large potentially inappropriate corrections.

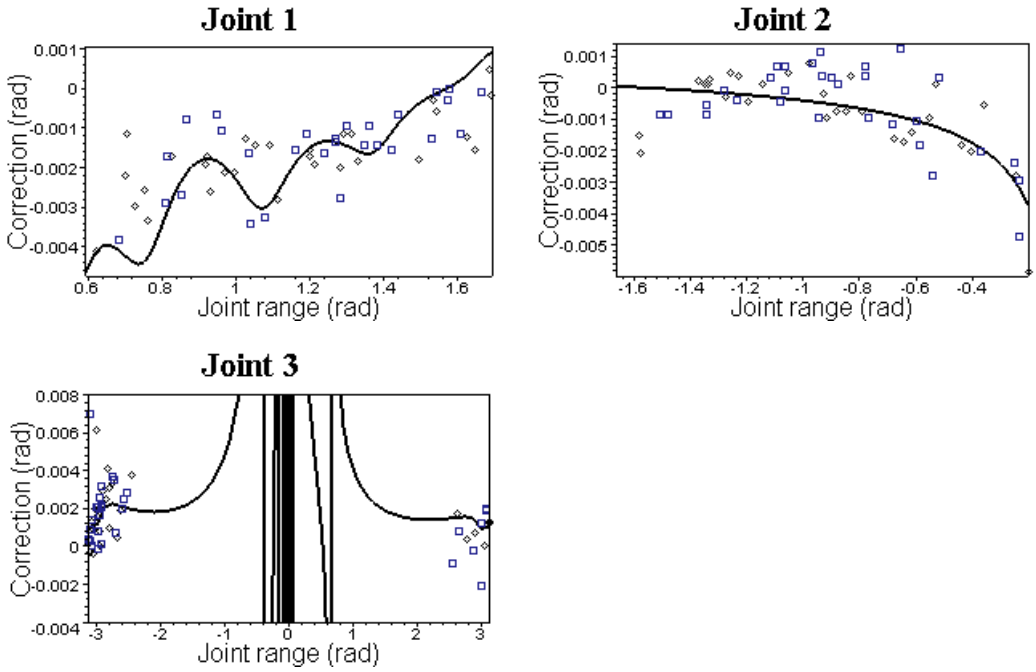


Figure 6-14: Evolved correction models for joint 1-3 plotted across the respective joint range along with calibrated joint angles (implicit targets) of the calibration set (boxes) and validation set (diamonds)

6.3 Experimental direct learning of joint correction models

As outlined in section 4.5 the second symbolic calibration method evolves the correction models for each joint independently based on the local error between nominal and calibrated joint configurations (see equation 4.12 on page 73). To implement this fitness measure the fitness evaluation (method *evaluate(...)* of class *gp_robot_chromosome* described in Table 5-6) has been appropriately adjusted. The

evolution of correction models for each joint within the corresponding GP system was controlled by the same parameters as in the experiments using distal supervised learning (see Table 6-1). However, the number of generations was limited to 200. The evolutionary model induction process was carried out sequentially[44] for each of the six joints. The reduction of the summed absolute joint error (see equation 4.12 on page 73) during a typical run of the calibration is illustrated in Figure 6-15.

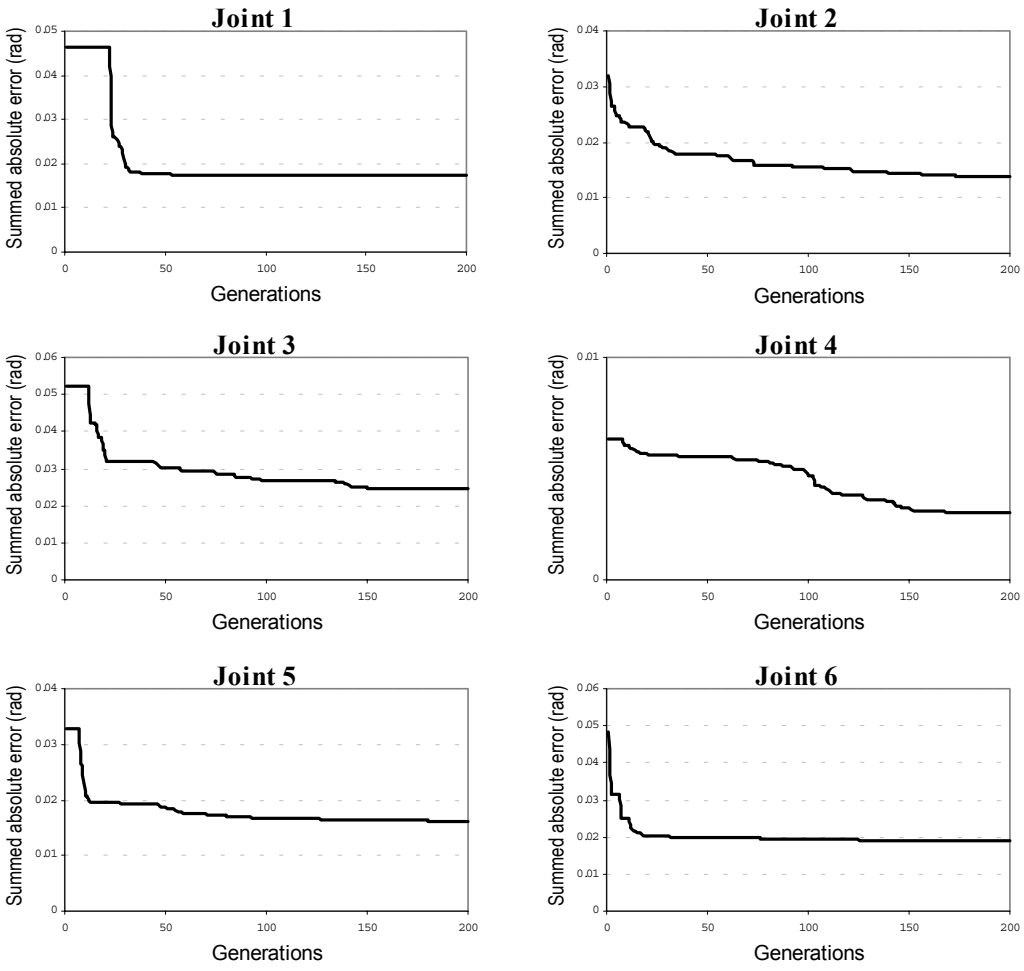


Figure 6-15: Summed absolute joint error being the fitness during the evolution of each individual correction model

The correction models evolved for each joint are listed in Table 6-4. Table 6-5 shows the results of the local joint error reduction for each joint and the resulting improvement of the absolute positional error of the robot tool.

[44] The evolution of the joint correction models could have been performed in parallel as indicated in section 4.5. The implementation of this concept however is beyond the scope of this research.

Joint	Evolved symbolic expression
1	SIN(0.8284485152745139*0.5474638508255257%theta*SIGN(0.2529984435560167) *(theta*theta*theta%SIGN(theta)-SIGN(COS(theta))*SIGN(theta))*(0.17336198 00408948*0.01056904812768944)%theta-SIN(SIN(SIN((0.1751177251503037* 0.010 74251533555101)%SIGN(0.4726706747642445))))*SIGN(0.9204992828150274))
2	(0.0490704367198706*COS((theta*theta*theta)%0.3623189642017884))%(SIN(SIN(th eta))*COS(theta)*SIN(0.545290505691702+theta)*(0.6641098513748589+theta)+0.26 94361857966857*0.03566191595202491%(0.6644795068208869+theta)+0.00489925 8400219733*0.84789574877163+SIN(theta))*0.3054859920041505*0.049348200933 8664*COS(theta%(COS(theta)*SQRTp(theta)*SQRTp(0.2692107150486771))*(SIN(0. 545290505691702+theta)*0.6639894253364665*0.6657387768181402)%(theta*theta *theta+SIN(0.6641098513748589+theta)))
3	SQRTp(SQRTp(0.543200460829493%(0.135572740867336%0.1914965361491745)))*0.135572740867336%(0.7125958891567735-SIGN(SIN(theta%0.48246711 63060 396)))* 0.1370563524277474*(0.7133252052369761-0.6703979155858028) %((SIN(SIN(0.7124685659352397-0.6691388286996063))-(0.7131533402508621- SIN(0.4481232490005188%0.7135958891567735)))%(theta*SIN(0.448103274636066 8%0.1914965361491745)*SIN(theta))-SIGN(SIN(theta%0.4824671163060396)))
4	(theta-(theta-0.03497595141453291)*(theta-0.03397595141453291)-SIN(theta)*theta-0.03318218024231696*COS((SIN(0.1224331492049928)+theta)* (0.0635253 1510361034%theta)%theta)-((0.03292576372569964*0.06352531510361034%theta-0.03440845973082675)%(0.2691939909054842-0.03497595141453291)-COS(COS (0.03424846644489883-theta)*0.7089775688955351%theta))* (0.1209600665303507 +theta)*COS(0.06152531510361034%theta*(0.06252531510361034%theta)%theta))*(theta-(theta-theta-0.03440845973082675)*SIGN(0.5189672536393323)-(SIN(0.122 0226142155217)+SIN(SIN(theta)))*COS(0.06152531510361034%theta*(0.062525315 10361034%theta)%theta))*theta
5	0.02016710409863582*(SQRTp(theta%(theta-theta)-theta*theta*theta) *0.8962777184362316*SIGN(theta)*0.3594566179387799-SQRTp(SQRTp(0.9496652729880674*theta-COS(theta))))*0.08997350993377484
6	SIN((theta+theta)*COS(0.3231803338724936-(theta-theta)))*COS(SQRTp(theta %0.6247010254219184-SIN(0.3126316110721152)))*((theta-(theta-theta)-0.306131 1685537279+0.3032620319223609-theta)%theta)%(COS(SIN((SIN(theta)+theta) *theta %0.6246228064821314*(0.3038389690847499-theta)))+SQRTp(SIN((theta-theta+SIN(theta))*SIN(COS(0.8384830011902219)))))

Table 6-4: Evolved symbolic expressions of joint correction models for joint 1-3 established using direct learning

		Prior to calibration	After calibration
Absolute mean joint error (rad)	Joint 1	1.54578E-03	5.81827E-04
	Joint 2	1.06465E-03	4.58385E-04
	Joint 3	1.73486E-03	8.22065E-04
	Joint 4	2.10844E-04	1.00212E-04
	Joint 5	1.09374E-03	5.38923E-04
	Joint 6	1.60570E-03	6.38020E-04
Mean positional tool end point error on calibration data set (mm)		1.730961	0.939094
Mean positional tool end point error on validation data set (mm)		1.850646	0.891515

Table 6-5: Calibration results using the correction models evolved by direct learning

Figure 6-16 and Figure 6-17 illustrate the improvement of the absolute positional error of the robot tool using the six joint correction models in Table 6-4 on poses from the calibration and validation data set respectively.

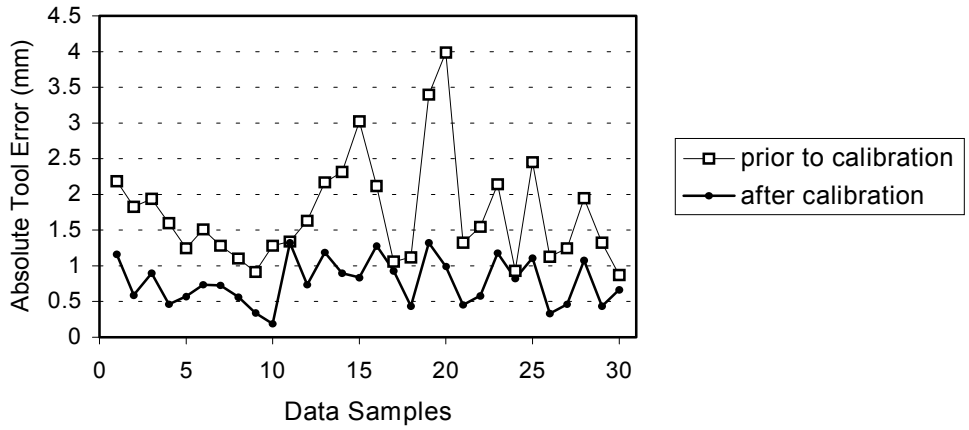


Figure 6-16: Comparison of the absolute positional error of the robot tool end point on the calibration data set prior and after calibration (direct learning)

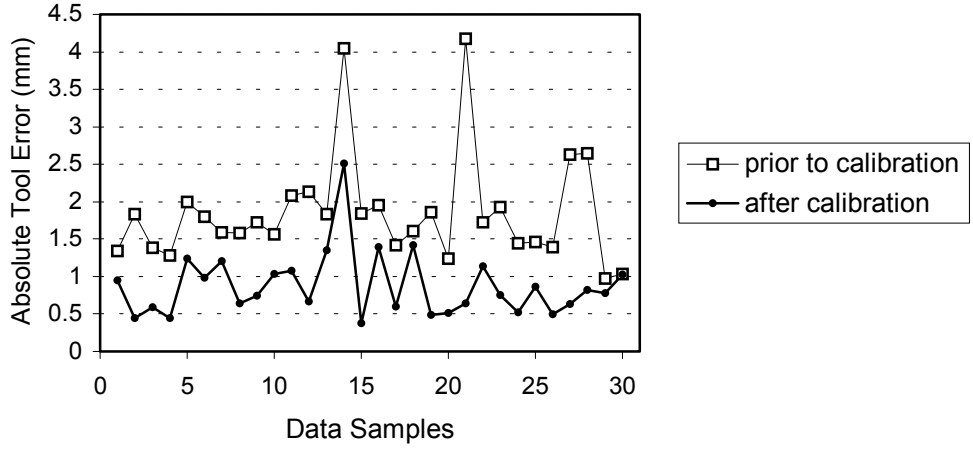


Figure 6-17: Comparison of the absolute positional error of the robot tool end point on the validation data set prior and after calibration (direct learning

The graphs of the six evolved correction models in Table 6-4 plotted across the range of their respective joints are shown in Figure 6-18. It is particularly worth noting that the plots of the symbolic expressions evolved for Joint 2 and 4 show a very close match between targeted and modelled corrections over a wide range. However, as with the results using distal supervised learning described in the

previous section, the GP algorithm evolved correction terms introducing discontinuities, which need to be evaluated in order to avoid undesired corrections. Since the same calibration data set was used as for the distal supervised learning experiment, the correction model for joint 3 was evolved based on data from two separate intervals as described in section 6.2.2.

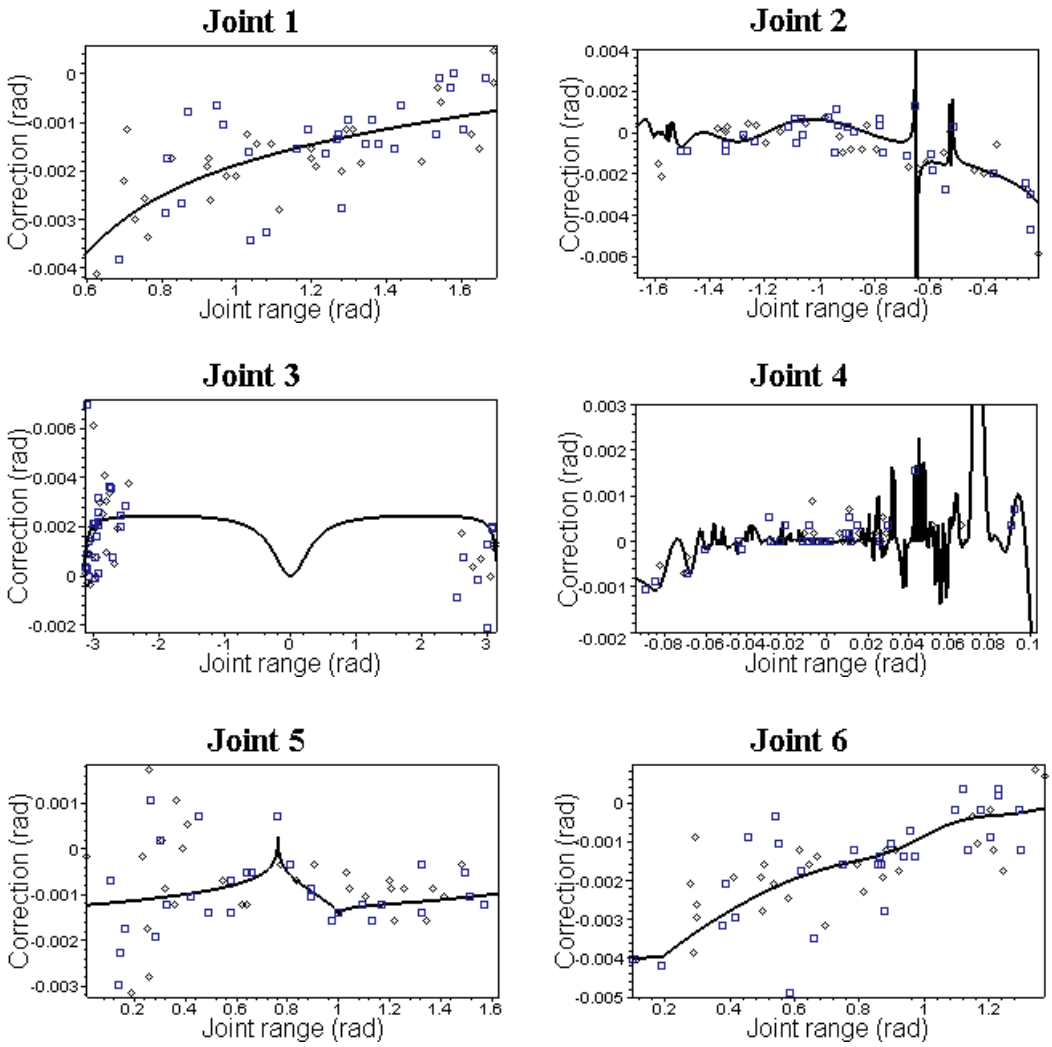


Figure 6-18: Evolved correction models (Table 6-4) for all six joint plotted across the respective joint range along with calibrated joint angles (explicit targets) of the calibration set (boxes) and validation set (diamonds)

6.4 Discussion

The results of the symbolic calibration of a PUMA 761 robot using both distal supervised learning and direct learning of correction models were taken from typical average runs of the calibration system developed in this research. That is, by using the GP parameters adopted for the experiments (Table 6-1) described in this chapter, similar reductions of the positional error of the robot tool may be produced, despite the stochastic search characteristic of genetic programming. From a set of terminals and primitive functions both methods generated complex correction models, which significantly reduced the absolute mean positional tool error by about 50% (see Table 6-3 and Table 6-5). Based on the calibration data used, the distal supervised learning method achieved this error reduction by inducing correction models for the joints 1-3 only. The direct joint error learning method enabled the induction of individual corrections for each of the 6 joints.

The quality of the evolved correction models depends among others on the chosen terminal and function set and the restrictions imposed by the GP parameters (Table 5-8). Many degrees of freedom are left to the experimentator to determine these GP parameters. The terminal and function set have both been chosen to contain components generally occurring in models capturing non-geometric effects (see section 2.2.2 for examples of non-geometric models). By introducing different, perhaps more complex functions to the function set than those used, the evolution could be accelerated or the accuracy of the corrections improved further. The GP parameters used for the experiments (see Table 6-1) have been accommodated to compromise structural complexity of the correction models, computation time required for calibration and evolutionary progress. Structural complexity is in this context related to the number of nodes in a symbolic expression and is limited by tree depth- related parameters. With a maximum tree depth of 9 (which yields maximal $\mathbf{2^{9+1} - 1}$ nodes to be evaluated for a balanced tree) and using the terminal/function sets shown in section 6.2 it is possible to generate symbolic expressions of any non-geometric model introduced in section 2.2.2 (except infinite Fourier series). Allowing more structural complexity by increasing the tree-depth parameter to 12 and 13 extremely degraded the evaluation performance but could not produce any better results in trials conducted. The size of a population of symbolic expressions

was in the experiments with both methods set to 300 as this number offers a large diversity of individuals and a reasonably efficient computation. Increasing the population size further to 500 and 1000 in evolutionary trials did not lead to better correction models but to significant losses of the calibration performance. The number of generations was limited to 1000 for distal supervised learning and to 200 for each of the 6 GP system instances in the calibration system when evolving correction models by direct learning. These numbers were chosen since average runs of a calibration conducted in the experiments using either method did not produce significant reduction of performance index (using distal supervised learning) respectively joint error (using direct joint error learning) beyond these generation limits. This finding however does not exclude the general potential of the calibration method to generate better performing correction models since the convergence speed in genetic programming as with all evolutionary computation methods is entirely unpredictable. An example for this unpredictability of the convergence speed is a calibration experiment conducted in this research using distal supervised learning in which the evolution was set to run over 10000 generations only to produce eventually a performance index of 49.39002. A reason for the little progress during a comparatively long evolutionary period in this particular example could have been code bloat (see e.g. [53]). This is a serious problem in GP and occurs particularly in later generation hampering the evolutionary progress by impairing the applicability of the evolutionary operators particularly subtree crossover but also subtree mutation.

As illustrated in Figure 6-7 and Figure 6-15 genetic programming usually progresses quickly in initial populations and slows down the convergence speed in the later course of evolution. This suggests that it is relatively easy for genetic programming to rapidly establish new populations that produce better performing individuals after a new fitness measure is applied (This is the case at initialisation where a ‘new’ fitness measure is applied assigning different values to randomly generated individuals). This suggestion is particularly supported by Figure 6-7, Figure 6-8 and Figure 6-9 documenting the evolutionary progress in the distal supervised learning approach. The progress in generating fitter individuals is much quicker after the calibration algorithm (Figure 4-6) switched to another joint (see Figure 6-9) to evolve its correction model. Due to the interdependence of the correction models in this co-evolutionary calibration scenario the fitnesses of the

models in the suspended populations change whenever a better performing correction model occurs in the active population. When the calibration algorithm switches to another joint (due to the higher potential of correcting the remaining error; see equation 4.11) its population becomes active. The fitness values of the correction models in this population will have changed compared to the time before this population has been suspended. Hence by evolving the calibration model in the active population the fitness measure for the suspended populations is implicitly redefined. For example at generation 336 the calibration algorithm switches from joint 1 to joint 3 (Figure 6-9). The last generation in which the population of joint 3 was active is generation 108. By evolving the correction model for joint 1 until generation 336 the fitness of the correction models in the suspended populations 2-5 is subject to change. When the algorithm switches to joint 1 at generation 336 the correction models in the population for joint 1 have different fitness values than in generation 108. Shortly after this "context switch" the GP system for this joint evolves a correction model that reduces the performance index. This behaviour of rapid performance index reductions after joint switches could be observed throughout the evolution (see Figure 6-7 and Figure 6-9). This property of the calibration could principally be exploited to increase the convergence speed of the performance index by introducing additional joint switches in periods with little evolutionary progress. However, this would mean to evolve correction models for joints with lower correction potential, which is generally possible as long as the measure (see equation 4.11) does not return a zero value for this particular joint (no correction of this joint would reduce the tool error). Eventually, the evolved joint correction models are more likely to cover errors that can be attributed to different joints. The potential of introducing additional joint switches and the effect it may have on the evolved correction models has not been investigated in the experiments carried out in this research and may be subject to further work.

An important issue for the calibration method presented in this work is the subsequent analysis of the evolved symbolic expressions to prevent undesired corrections being performed. Since symbolic regression constructs the correction models from a limited number of calibration samples (snap shot view) the validity of these models needs to be evaluated throughout the joint range defined by the calibration area. Fortunately, due to the protected definition of hazardous functions

(see section 3.3.1) used for symbolic regression it is guaranteed that corrections will be well defined throughout, even though perhaps not continuously (discontinuities were introduced by e.g. the SIGN function, which is used to model effects such as gear backlash). Undesired effects such as local oscillatory behaviour of the correction model potentially producing large values (as shown in Figure 6-14) need to be accounted for by introducing local damping or scaling, for example. The same applies to large corrections produced by divisions involving small denominator values.

Chapter 7

Conclusion and Outlook

In this work a novel technique for generating a robot calibration model for use in off-line programming was developed: the symbolic calibration method. This method is based on genetic programming or more specifically symbolic regression, which evolves joint correction models. These joint correction models are then used to reduce the positional error of the robot tool. Thus positional calibration of the robot is performed in joint space by modelling the error of the nominal inverse kinematic model, which is implemented in the robot controller.

The advantage of the developed method is the automatic generation of the correction models without presumptions about model structure or parameter values. In contrast, classical calibration methods (see [12][27][37][43][70]) require human involvement to establish a calibration model (model based or approximation), which is subsequently fitted to calibration data employing methods from numerical analysis. As outlined in Chapter 2 indirect numerical methods such as gradient search can cause problems if the calibration model has parameter redundancies, parameter discontinuities etc. Also, the convergence against a globally optimal solution is with these conventional methods not guaranteed.

The symbolic calibration method developed in this work was implemented using both distal supervised learning (co-evolutionary approach), and direct joint error learning approaches. It combines the processes of automatically generating and evaluating correction models in a direct evolutionary search method based on

symbolic regression. By utilising the underlying concept of stochastic inference[45], symbolic regression has the potential to solve the calibration problem by finding a suitable or even the true structure and parameter values of a calibration model. Since the method is not confined to fixed model structures the proposed calibration method is more flexible than classical numerical calibration techniques.

The experimental implementation of the method developed in this work has demonstrated the potential genetic programming has to solve the static kinematic calibration problem. The results of calibration trials in Chapter 6 show that genetic programming in both developed methods described in Chapter 4 is capable of reducing the positional mean absolute error of the robot tool by about 50% (55% for distal supervised learning). These error reductions were obtained from average runs of the calibration system and are, despite the stochastic property of the method, reproducible (using the same GP parameter configuration), even though the evolved correction models in different calibrations may vary in structural complexity. However, the calibration method is currently limited by the intrinsic problems inherited from the genetic programming paradigm, which mainly are the consumption of computational resources particularly computation time due to the general tendency of GP to reduce the convergence speed in later generations, and the unpredictability of the evolutionary progress. Due to these limitations a wider range of values of parameters that control the GP run could not be exploited in the scope of this work and remains subject to further investigation.

7.1 Suggestions for further work

The symbolic calibration approach developed in this work offers many opportunities for further investigations. The following suggestions are given:

(i) Enhancing speed of evolution by introducing parallel processing
(ii) Implementation of advanced evolutionary principles
(iii) Further mathematical analysis of the evolved correction models

[45] As John Koza [49] puts it: “knowledge derived logically from known facts is not new and therefore not patentable”. Stochastic implies unpredictability, which is both, potential and limitation of evolutionary computation.

(iv) Enhancing GP parameter study

(v) Subsequent numerical optimisation of the correction models

(i) On the technical side performance issues need to be discussed further to enable a more comprehensive parameter study and include more data samples for calibration. For example the calibration system used in this work could be enhanced in future experiments by implementing parallel distributed evolution of correction models for each joint using the direct joint error learning method described in section 4.5.

(ii) Even though the distal supervised learning method does not directly permit parallel evolution of the correction models due to their interdependence (see section 4.4.1), implicit parallel processing within a population could be performed (for one joint at the time) by introducing concepts of *niching* or *demetic grouping* (see e.g. [49]). This would split a population into demes ("islands"), which independently evolve their correction models (for the same joint!) based on a possible different GP parameterisation. An advantage of this isolation of individuals is that evolution can specialise in several areas in the search space. Unlike niching, demetic grouping introduces the migration concept, which allows fit individuals to move into other demes, where they might contribute useful genetic material. In fact, migration of correction models could also be introduced between the populations (between different joints) of the calibration system. This concept would enable population interaction by sharing genetic material from good performing correction models between otherwise isolated populations. For example non-geometric effects occurring in the joints may be described by similar correction terms (i.e. same structure using different coefficients). Therefore advanced GP concepts such as automatic defined functions (ADF) (see [50]) could be investigated in this context.

(iii) Making genetic material of good performing individuals available for sharing between other individuals across the population also relates to the building block hypothesis, which is controversially discussed in the GP community (see e.g. [16][53]). Investigations could be carried out to find and examine expressions commonly used in good performing individuals. The foundation for those future experiments is laid by the shared GP tree implementation described in Chapter 5 (and

illustrated in Figure 5-1) as this representation of the symbolic expression enables the identification of commonly used sub expressions.

(iv) The GP parameter study could be extended by including different, perhaps more complex functions in the non-terminal set. In addition the calibration could be performed using different values for GP parameters (e.g. population size, or permitting more structural complexity by increasing the GP tree depth) to further explore the potential of the calibration method.

(v) Furthermore the evolution could be enhanced by numerically fitting the generated correction models (i.e. fine-tuning of the values of the constant nodes). During the evolution a number of selected models could be numerically fitted to the calibration data set in an attempt to increase their accuracy. Those fitted individuals could then be directly carried over to the following generation (Lamarckian learning [5][52]). Alternatively, the by numerical fitting reduced performance index respectively joint error could be used as the new fitness value for the original model (prior numerical fitting). In this way the learning capability (capability to numerically fit to calibration data) of that model would be propagated rather than the fitted expression (Baldwin effect [5]). Technically, the assumption for numerical optimisation would be that the nodes generated by the ephemeral constant are treated as parameter nodes with the node value being the start value for the optimisation. However, prior to numerical optimisation the evolved models may have to be algebraically simplified/normalised. This is in most cases necessary since genetic programming typically tends to generate over-specified correction models. These models may become structurally very complex and contain an unnecessary large number of constant nodes. When treated as parameter nodes there are likely to occur redundancies or dependencies between these nodes, which would make fine-tuning of their numerical values difficult or impossible. Therefore using symbolic algebra and compiler methods such as constant folding [3] (replacement of constant expression by their value), constant propagation or term collection the structural complexity of the evolved expressions may be significantly reduced enabling subsequent numerical fitting. However, numerical fitting is suggested to be performed using direct methods (such as the Nelder-Mead simplex method [2]) rather than indirect gradient based methods, as the evolved expressions will most likely contain discontinuities (as encountered in the experiments), which will cause

gradient methods to fail. As algebraic simplification and both, direct and indirect optimisation methods are, however, computationally very time consuming the examination of Baldwin effect and Lamarckian learning will either be confined to a limited number of correction models in a population, or remains to be carried out with more powerful computer hardware becoming available.

Appendix A

A.1 The Robotrak measurement system

Robotrak® is a cable driven measurement system capable of recording robot end-effector positions. It consists of three measurement units *A, B* and *C* arranged in a planar triangle as illustrated in Figure A-1. Each unit provides a cord one end of which is attached to the robot tool *R* the other wrapped around a drum within the unit. The length of each chord (distance from the respective measurement unit: $\boldsymbol{r}_a$, $\boldsymbol{r}_b$ and $\boldsymbol{r}_c$ to the robot tool) is measured by the respective measurement unit based on incremental encoders, which generate pulses on rotation changes of the drum.

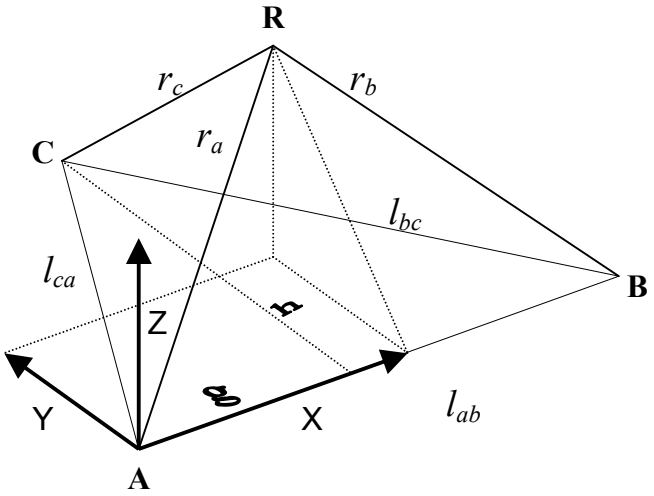


Figure A-1: Robotrak® geometry

The cord measurements from the tree units are transformed into a Cartesian frame using data from the geometry of the measurement system. The origin of the Robotrak co-ordinate system is chosen to be located at point *A* with the local *X*- Axis being aligned to l_{ab} and the local *Z*- Axis being perpendicular to the plane defined by the points *A*, *B* and *C*. The points *A*, *B* and *C* are thought to be centre points of spheres with the radii r_a, r_b and r_c respectively, for which the following relations hold:

$$r_a{}^2 = x^2 + y^2 + z^2, \tag{A.1}$$
$$r_b{}^2 = (l_{ab} - x)^2 + y^2 + z^2, \tag{A.2}$$
$$r_c{}^2 = (x - g)^2 + (h - y)^2 + z^2. \tag{A.3}$$

Solving the trigonometric identity $r_a{}^2 - x^2 = r_b{}^2 - (l_{ab} - x)^2$ for x yields:

$$x = \frac{r_a{}^2 - r_b{}^2 + l_{ab}{}^2}{2l_{ab}}.$$

Similarly, for the triangle *ABC*

$$g = \frac{l_{ca}{}^2 - l_{bc}{}^2 + l_{ab}{}^2}{2l_{ab}},$$

$$h = \sqrt{l_{ca}{}^2 - g^2}.$$

Eliminating z^2 in (A.3) using (A.1) and solving the obtained equation for y gives:

$$y = \frac{r_a{}^2 - r_c{}^2 - 2gx + g^2 + h^2}{2h},$$

$$\text{finally } z = \sqrt{r_a{}^2 - x^2 - y^2}.$$

The encoders of the three measurement units were connected to a MITC 12 interface card plugged into a PC. The contents of the actual encoder count registers (measure for the cord length) on the card could then be read by computer applications. Initially, Workspace [82] was used to gather measurement data. The usage of Workspace however proved difficult particularly when recording large amounts of data in several sets since the module for automatic data collection appeared to have programming errors (in version 4.0 used). The manual recording of data was not an option due to the amount of measurements taken and potential errors that might have been introduced (waiting for the robot to settle and confirm measurement). Therefore an application has been developed (and made available)

that enables an easy set-up of Robotrak and permits a convenient automatic data collection (Figure A-2). This application implements the transformation algorithm described in this section and outputs measured poses (x, y, z) listed in a file to be processed by the calibration system.

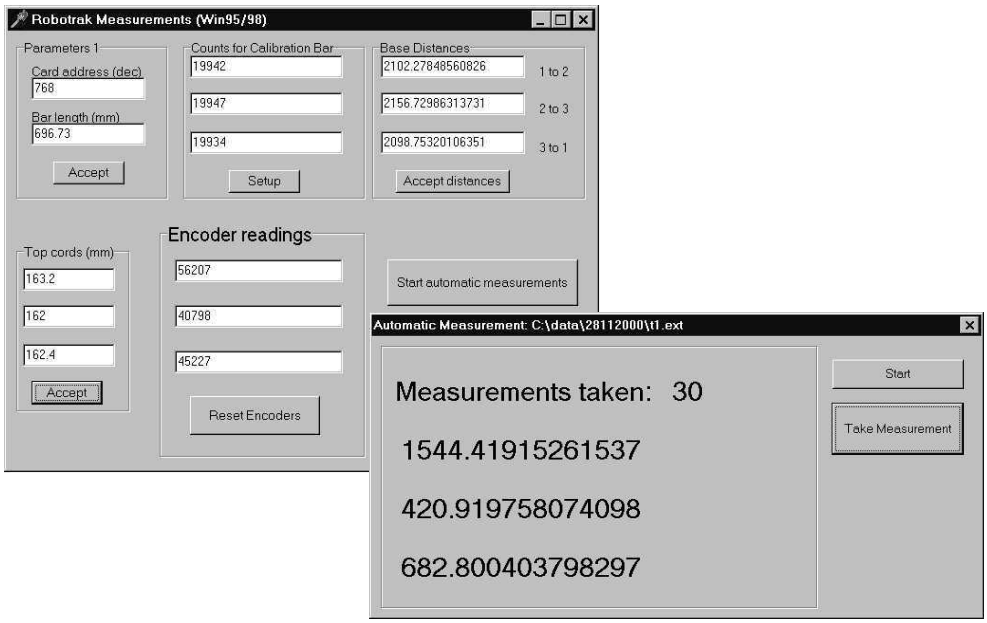


Figure A-2: Developed data collection application

Figure A-3: Laboratory arrangements

The set-up procedure for the Robotrak device is described as follows: First the cords need to be exercised in order to guarantee a good repeatability (cords must be tightly wrapped around the drums within the measurement units). This is performed

by repeatedly pulling out each cord for approximately two meters from the encoder base. The second step is the calibration of each encoder. A calibration bar is used to determine the number of encoder counts for a defined length. After this calibration the encoders are used to measure the distances between them: *A* for distance *AB*, *B* for distance *BC*, and *C* for distance *CA*. Then the lengths of the top cords (see Figure 6-1) attached the tool are to be measured. The cords of the measurement units are attached to the corresponding top cord together constituting the effective length from the robot tool to the respective measurement unit ($\boldsymbol{r}_a$, $\boldsymbol{r}_b$ and $\boldsymbol{r}_c$ respectively).

The accuracy of the particular Robotrak system used in this work was estimated in previous experiments at 0.27 ± 0.21 mm [45].

A.1.1 Local frames

For the evaluation of accuracy both nominal robot poses and measured data delivered by the Robotrak measurement system must be expressed relative to the same co-ordinate frame. However, the transformation from Robotrak frame to robot base frame is usually not given exactly, in fact, it is generally identified simultaneously with the other kinematic parameters during calibration [12]. Alternatively this transformation may be accomplished by establishing a local frame that is used to transform data from the Robotrak system frame to the robot base frame. This method has been adopted in this work and in [45].

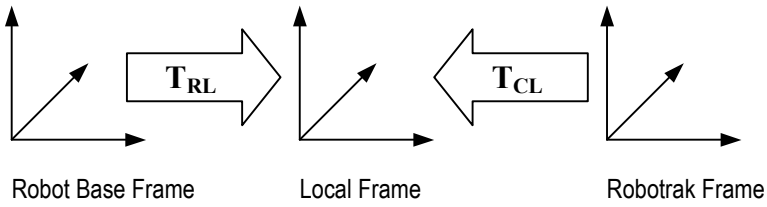


Figure A-4: Local frame transformations

A local frame is defined within the robot workspace by 3 end-effector poses ($\mathbf{L}_1$, $\mathbf{L}_2$ and $\mathbf{L}_3$) the positional co-ordinates $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ of which are recorded relatively to the robot base frame and the Robotrak frame.

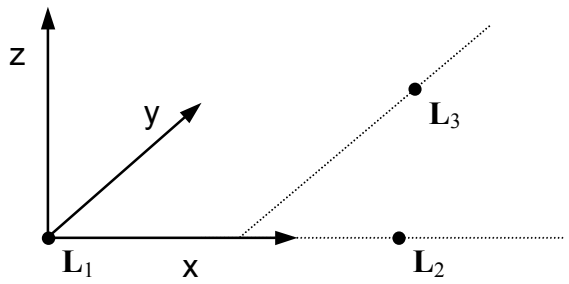


Figure A-5: Local frame definition

The local x-axis passes through $\mathbf{L}_1$, and $\mathbf{L}_2$. The local y- axis is defined by being perpendicular to the x-axis and passing through $\mathbf{L}_3$. The origin of the local frame lies in $\mathbf{L}_1$. This local co-ordinate frame is represented by a homogenous matrix $\begin{bmatrix} \mathbf{R} & \mathbf{L}_1 \\ \mathbf{0}^{\mathrm{T}} & \mathbf{1} \end{bmatrix}$ with $\mathbf{R}$ being the rotation matrix the normalised column vectors of which represent the local frame axes (X,Y,Z).

Having established the local frame matrix $\mathbf{T}_{\mathrm{RL}}$ relative to robot base, robot poses $\mathbf{P}_{\mathrm{RF}}$ can be represented in local frame co-ordinates by:

$$\mathbf{P}_{\mathrm{LF}} := {\mathbf{T}_{\mathrm{RL}}}^{-1}\mathbf{P}_{\mathrm{RF}}.$$

Analogously, measurement data $\mathbf{P}_{\mathrm{CF}}$ relative to the Robotrak frame can represented relatively to the local frame as $\mathbf{P}_{\mathrm{CLF}}$ by:

$$\mathbf{P}_{\mathrm{CLF}} := {\mathbf{T}_{\mathrm{CL}}}^{-1}\mathbf{P}_{\mathrm{CF}}$$

where $\mathbf{T}_{\mathrm{CL}}$ is the local frame matrix relative to the Robotrak frame.

Hence measurement data relative to the Robotrak frame can be represented in terms of robot base frame co-ordinates by:

$$\mathbf{P}_{\mathrm{RF}} := \mathbf{T}_{\mathrm{RL}}{\mathbf{T}_{\mathrm{CL}}}^{-1}\mathbf{P}_{\mathrm{CF}}.$$

A.2 Denavit- Hartenberg parameters

The Denavit- Hartenberg parameterisation describes the relative displacement of two consecutive link frames using 4 parameters for the i^{th} link.

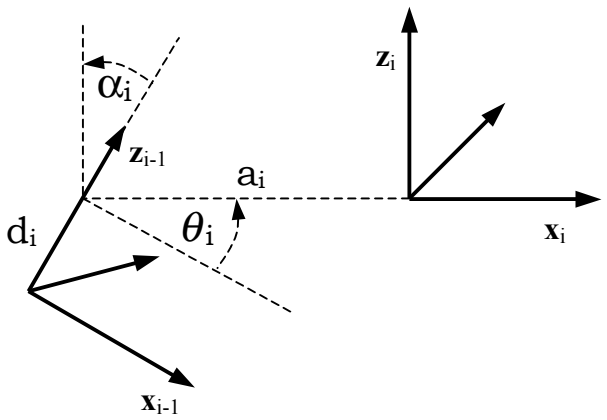


Figure A-6: Denavit-Hartenberg parameters

The parameter α_i is the angle between joint axis $\mathbf{z}_{i-1}$ to $\mathbf{z}_i$ about $\mathbf{x}_i$, θ_i the angle between $\mathbf{x}_{i-1}$ to $\mathbf{x}_i$ about joint axis $\mathbf{z}_{i-1}$ (θ_i joint variable for rotary joints), $\mathbf{a_i}$ is the distance from joint axis $\mathbf{z}_i$ to $\mathbf{z}_{i-1}$ along $\mathbf{x}_i$ and $\mathbf{d_i}$ the distance from $\mathbf{x}_i$ to $\mathbf{x}_{i-1}$ along joint axis $\mathbf{z}_{i-1}$ ($\mathbf{d_i}$ is joint variable for prismatic joints).

The displacement of the i^{th} link frame to the previous link frame in terms of orientation and position is the product of consecutive elementary homogenous rotation and translation matrices resulting in the following homogenous matrix:

$$\mathbf{A_i} = \mathbf{Rot_z}(\theta_i)\mathbf{Trans_z}(\mathbf{d_i})\mathbf{Trans_x}(\mathbf{a_i})\mathbf{Rot_x}(\alpha_i)$$

$$= \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cdot\cos(\alpha_i) & \sin(\theta_i)\cdot\sin(\alpha_i) & \mathbf{a_i}\cdot\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cdot\cos(\alpha_i) & -\cos(\theta_i)\cdot\sin(\alpha_i) & \mathbf{a_i}\cdot\sin(\theta_i) \\ \mathbf{0} & \sin(\alpha_i) & \cos(\alpha_i) & \mathbf{d_i} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

The co-ordinate frame of the tool of a 6-link manipulator relative to the robot base frame can then be expressed as:

$$\mathbf{T_A} = \prod_{i=1}^{6} \mathbf{A_i} = \begin{bmatrix} \mathbf{Tr} & \mathbf{Tp} \\ \mathbf{0}^T & \mathbf{1} \end{bmatrix}$$

with $\mathbf{Tr}$ being the 3×3 rotation matrix expressing tool orientation and $\mathbf{Tp}$ the 3×1 position vector. In order to obtain the position of the tool tip the matrix $\mathbf{T_A}$ is multiplied with the tool transformation matrix $\begin{bmatrix} \mathbf{I} & \mathbf{X} \\ \mathbf{0}^{\mathrm{T}} & \mathbf{1} \end{bmatrix}$ where $\mathbf{I}$ is the 3×3 identity matrix and $\mathbf{X}$ the 3×1 vector which expresses the translation from the origin of the tool frame to the tool tip.

The Denavit-Hartenberg parameters of the PUMA 761 manipulator used in this work are given by [75] as:

Link	**α**	**a** (mm)	**d** (mm)
1	$-\pi/2$	0	0
2	0	650	191
3	$\pi/2$	0	0
4	$-\pi/2$	0	600
5	$\pi/2$	0	0
6	0	0	125

Table A-1: DH parameters of the PUMA 761 manipulator

Appendix B

B.1 Publications

Dolinsky J.-U., G. J. Colquhoun and I. D. Jenkinson. (1998). *A comparison of techniques for modelling robot dynamics*: Proceedings of the 14th national conference on manufacturing research, University of Derby, UK.

Dolinsky J.-U., G. J. Colquhoun and I. D. Jenkinson. (2000). *Structural identification and calibration of kinematic robot models by genetic search*. Proceedings of the 33rd international MATADOR conference, University of Manchester, Institute for Science and Technology (UMIST), UK.

Structural identification and calibration of kinematic robot models by genetic search

Jens- Uwe Dolinsky, I.D.Jenkinson, G.J. Colquhoun
Liverpool John Moores University, School of Engineering,
Byrom Street, Liverpool L3 3AF
Email: {engjdoli, I.D.Jenkinson, G.J.Colquhoun}@livjm.ac.uk

Abstract: Accurate robot modelling is of great importance to the application of enhanced robot programming tools such as Offline Programming systems. This paper describes a prototype of an automated kinematic modelling environment, which is primarily based on evolutionary computation. A genetic algorithm herein attempts to find an optimal model structure of the forward kinematic of an industrial robot based on measurements reflecting individual characteristics. Finally it will be reported on results obtained from simulation experiments.

1 Introduction

Modern robot programming methods such as Offline Programming require accurate robot models sufficiently capturing the robot kinematics. Since every robot, even within the same series, has individual deviations of kinematic properties which is due to e.g. manufacturing tolerances and tear, the controller model has to be updated e.g. by calibrating its kinematic parameters with measurement data taken from that specific robot. In Offline Programming robot programs are developed and validated within a simulated environment assuming an accurate match of robot and workcell models with their physical counterparts.

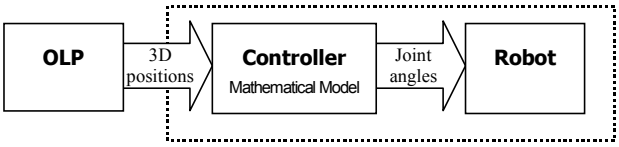


Figure 1: Relations OLP ↔ physical robot

Since information about the robot controller model is usually not available (the black box in Figure 1) and in order to create portable programs the common approach of an Offline Programming System (OLP) such as Workspace [5] to achieve higher absolute accuracy of a particular robot is to employ path error compensation. A calibrated kinematic model acts as a filter, which alters target positions in the robot program to direct the robot to the desired targets.

Kinematic Calibration in robotics is a well-established field of research that has delivered several models and calibration methods addressing special types of robots and numerical stability issues of the parameter identification procedure [2]. Some approaches utilise black box models based on artificial neural networks (ANN) with little or no relation of the connection weights used to physical robot parameter. In general the approach has been to determine a fixed model structure and to subsequently fit its parameters using measurement data.

This paper presents a new automated general hybrid modelling method to generate a forward kinematic robot model based primarily on evolutionary computation and gradient search. The genetic algorithm proposed has the task to create an appropriate model structure whose parameter will be numerically identified by subsequent gradient search. Unlike other hybrid search heuristics, e.g. genetic algorithms on artificial neural networks, the objective of this approach is not to derive a black box model. Instead knowledge of the nominal kinematic model, e.g. the serial links and their order, will be integrated and preserved.

2 Genetic modelling of robot kinematics

Genetic Algorithms (GA) are search procedures inspired by concepts of natural evolution. In principle a population of potential solutions (termed individuals) is searched or explored for the best performing or fittest individuals. To keep and exploit their characteristics those individuals will be selected for reproduction or recombination in further generations eventually undergoing mutational changes. In that way over a number of generations the algorithm attempts to gradually breed highly fit solutions to a problem. An introduction to GA's can be found in [1]. Genetic Algorithms have been found to be useful for solving many different types of problems across different disciplines. This independence from the actual problem domain and the underlying concepts, which are relatively easy to implement in computer programs, has contributed to their success.

2.1 Model representation

Kinematic Modelling of robots can be divided into geometric and non-geometric modelling. Pure geometric models such as the Denavit-Hartenberg (DH) (see e.g. [3]) model, merely cover the geometric properties e.g. link lengths and angles between neighbouring axes. Non-geometric properties e.g. gear compliance and gear backlash however contribute considerably to the accuracy of a robot and need to be taken into account (see e.g. [4]).

In this approach a genetic algorithm attempts to model the influence of non-geometric properties by inserting additional co-ordinate transformations into a DH model. The fact that forward kinematic models can be built from sequences of elementary homogenous transformations makes them an appealing representation for genetic algorithms permitting effective application of genetic operations such as crossover or mutation. The principle idea is to view the entire kinematic robot model as an ordered sequential concatenation of fundamental homogenous transformations (DH- transformations with nominal parameter values provided by the robot manufacturer) together with additional co-ordinate transformations (such as translations along and rotations about the x, y and z-axis) inserted by the genetic algorithm. Those sequences make up the kinematic genome[46] on which the genetic algorithm operates. Figure 2 shows the genome of the 6-link PUMA 762 robot used in our experiments.

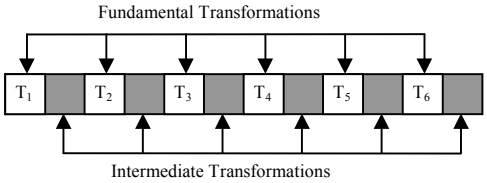


Figure 2: Kinematic Genome

The initial population of genomes is seeded with each genome containing the fundamental transformations (T_1 to T_6) in the order according to the links of the robot. The genome of the kinematic model in Figure 2 consists of 12 gene sections, 6 of which (the fundamental transformations) must not be altered. This creates a primary schema, which expresses the initial knowledge (that of the provided serial link design and specifications) which is to be preserved. Random numbers of different additional transformations will be placed between these fundamental transformations by the genetic algorithm. The inserted transformations can be single translations, rotations or composed transformations (e.g. building blocks).

[46] Genome or chromosome (string of genes) in this context means the entire information required to construct a kinematic model

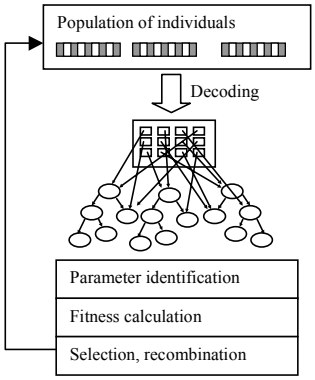


Figure 3: Genetic Algorithm

Thus the genetic algorithm explores the space of sequential homogenous transformations for the optimal kinematic design of a particular robot. It evaluates, selects and recombines the transformation strings (genomes) within a population of a given size.

2.2 Decoding and evaluation

In order to evaluate a population of models, all genomes have to be decoded and prepared for evaluation. For each genome the corresponding homogenous transformation matrix (phenotype) is created. Each gene in the section of intermediate transformations represents one single transformation for which a parameter needs to be generated. Finally all transformation matrices are multiplied symbolically to form the computational model represented by its homogenous transformation matrix (Figure 3). The internal representation of this transformation matrix is an array of symbolic expressions (binary trees) instantiating the equations for position and orientation. Note that since all transformations are homogenous the genetic algorithm, although performing drastic structure changes, cannot violate the orthonormality constraint of the rotation matrices. Also, the application of an analogous crossover technique prevents the genetic algorithm from changing the order of the fundamental transformations, which preserves the predefined serial link design. This means only genes within the same gene section can be exchanged.

It is possible during the course of evolution that the algorithm creates gene sections with several consecutive equal transformations e.g. 2 translations along the x- axis. This would obviously cause a parameter redundancy, which is avoided immediately during decoding by ignoring those multiple equal transformations. However, those redundancies are kept in the genome in order to preserve a diversity of genetic material for further generations.

2.3 Parameter identification

Once the model has been instantiated, its parameters need to be identified using measurement data. The forward kinematic model can be written as:

$$\mathbf{y} = \mathbf{f}(\theta, \phi), \tag{2.1}$$

where $\mathbf{f}$ returns the position and orientation of the end-effector tool depending on the given joint angle set θ and parameter configuration ϕ. Identification is carried out by minimising:

$$\sum_{\mathbf{i}=1}^{\mathbf{n}} \| \mathbf{y_i} - \mathbf{f}(\theta_\mathbf{i}, \phi) \|^2 \tag{2.2}$$

with subject to ϕ, where $\mathbf{y_i}$ is the $\mathbf{i}^{\text{th}}$ measurement and $\mathbf{n}$ is the number of measurements. Since the model equations are nonlinear in rotational parameters the identification has to be carried out iteratively e.g. by nonlinear least squares. For this the model equations have to linearised by first order Taylor expansion around the current parameter estimate. The functional (2.2) to be minimised can then be rewritten as:

$$\| \triangle\mathbf{y} - \mathbf{C}\triangle\phi \|^2 \qquad \text{with } \mathbf{C} = \begin{bmatrix} \mathbf{C_1} \\ \vdots \\ \mathbf{C_n} \end{bmatrix} \text{ and } \triangle\mathbf{y} = \begin{bmatrix} \triangle\mathbf{y_1} \\ \vdots \\ \triangle\mathbf{y_n} \end{bmatrix} \tag{2.3}$$

subject to $\triangle\phi$ and $\mathbf{C}$ being the Jacobian of $\mathbf{f}$ approximated by finite differences at the current parameter estimate. To solve this linear minimisation problem for the parameter update (Gauss-Newton update) vector yields:

$$\triangle\phi = (\mathbf{C^T C})^{-1}\mathbf{C^T}\triangle\mathbf{y} \tag{2.4}$$

Hence the parameter values are iteratively obtained by:

$$\phi_{\mathbf{k+1}} = \phi_{\mathbf{k}} + \triangle\phi_{\mathbf{k}} \tag{2.5}$$

The Moore-Penrose generalised inverse $(\mathbf{C^T C})^{-1}\mathbf{C^T}$ is not formed explicitly in order to avoid numerical instabilities e.g. due to round-off errors. Instead orthogonal decomposition is applied to $\mathbf{C}$ in (2.3) by Householder Transformations. However $\mathbf{C^T C}$ may become singular and hence not invertible. In that case the algorithm uses the Levenberg-Marquardt update $\triangle\phi = (\lambda\mathbf{I} + \mathbf{C^T C})^{-1}\mathbf{C^T}\triangle\mathbf{y}$ with λ being a positive scalar constant to be determined and $\mathbf{I}$ being the identity matrix.

2.4 Fitness calculation

The fitness value of an identified model is the residual error (2.2) computed on a second set of measurements. Thus the fitness is a function of the evolved kinematic structure and the identified parameters. The genetic algorithm exploits the Baldwin effect e.g. the learned information (identified parameter values) only affects the fitness function and is not backcoded or retained in the genetic description of an individual model.

3 Experiment

For our experiments we implemented a steady state GA[47]. Individuals (kinematic models) are selected from the population by Tournament Selection (see [1]). This type of selection was chosen because it is easy to implement, and by setting an appropriate tournament size it allows convenient adjustment of the selective pressure and thus the convergence property of the genetic algorithm. To create a new model, two tournaments were run and the best performing model of each was chosen to be parents for the new model.

The initial population was seeded with genomes with each intermediate gene section containing up to 5 random elementary transformations. Fundamental transformations were initialised with the nominal DH parameters of the PUMA 762 robot. The parameters of elementary transformations in the intermediate gene sections were initialised with the value of 0.01 (translational and rotational parameter). The number of measurements used for identification and evaluation was 50. The probability of applying crossover was 0.8 and mutation 0.3. The evaluation of 50 generations (population size was 15) took about 20 min on a PC (with a 500 Mhz Intel Celeron processor) running Linux. The results of a typical run of the genetic algorithm are shown in Figure 4.

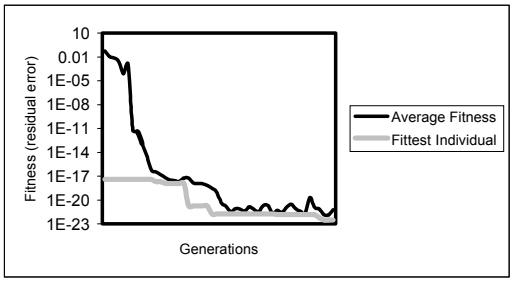


Figure 4: Results obtained from a GA run over 50 generations (population size: 15)

As the evolution progresses, fitter individuals appear in the population, the fitness of individuals improved almost gradually. However, because of the limited population size of 15 individuals the GA settles relatively quickly. To keep the evolution going i.e. to explore other genome configurations additional mutation was applied to some individuals in generations, the spikes in the average fitness curve (Figure 4) illustrate this effect. This method proved useful because this perturbed genetic material contributed to slightly fitter individuals.

[47] Offspring created by the GA replaces bad or worst performing individual in a population.

4 Conclusion

The genetic algorithm presented appeared to be well capable of fitting a kinematic robot model (structure and parameter) to measured data. Despite compromises made with the choice of the genetic algorithm parameter (e.g. number of generations and population size) due to the high computational complexity, it performed robustly and delivered parametric models, which performed even better on our measured data than an ordinary DH model calibrated with these data. The algorithm works with simple mechanisms (e.g. estimation of parameter values), which need to be refined for further support of the results. Further work planned in this context is to reduce the number of parameters to be estimated by applying variable projection, which requires an initial estimate for nonlinear parameters only.

References

[1] Beasley, D., D.R. Bull and R.R. Martin, (1993), *An Overview Of Genetic Algorithms*. University Computing, Part 1 "Fundamentals", 15(2) 58-69, Part 2 "Research Topics", 15(4) 170-181.
[2] Bernhardt, R. and S.L. Albright, editors. (1993). *Robot Calibration*. Chapman & Hall, London.
[3] Hollerbach, J.M., (1999), *Introduction to Robotics*, Lecture Notes CS 5310/6310 & ME 5220/6220, Chapter 4, University of Utah.
[4] Whitney, D.E., Lozinski, C.A. and Rourke, J.M., (1986), *Industrial Robot Forward Calibration Method and Results*, ASME Journal of Dynamic Systems, Measurement and Control, Vol. 108, No.1.
[5] Workspace 4, (1998), User Manual Robot Simulations Ltd, Newcastle Upon Tyne, UK.

A comparison of techniques for modelling robot dynamics

J.U.Dolinsky, G.Colquhoun, and I.D.Jenkinson
School of Engineering
Liverpool John Moores University
Byrom Street, Liverpool L3 3AF
Tel. 0151 231 2081

An industrial robot can be programmed in two ways either, by writing the program online using teach programming or by generating the program off-line with the aid of a CAD based Off- Line Programming system (OLP).
An OLP uses simulation to imitate the robot activities. This simulation is based on a model (for computing a mathematical model) which sufficiently describes the robot kinematics, dynamics and the controller strategy. In general, analytic models are used. However, some investigators have proposed models based on neural networks.

In this paper the advantages and disadvantages of neural network based models are explained and compared with general analytical modelling methods.
Furthermore possibilities for the refinement and adaptation of the analytical model of the respective robot are discussed. This includes investigations regarding the applicability of automatic mechanisms for parameter identification etc. in order to automate model generation.

1 Introduction

Offline Programming Systems allow the development and test of robot programs without seizing the physical robot. With a CAD system a sequence of tasks is designed (3D data) and thereafter applied to a simulator, which bases on kinematic, dynamic as well as the controller model of the respective robot.

In general the problem a controller of an industrial robot has to solve, may be simplified and expressed by the following formal notation:

$$T = f\left(X_{Start}, X_{Target}\right)$$

Between two given points X_{Start} and X_{Target}, which are vectors in the 3 dimensional cartesian space, a trajectory T (a series of configurations → end effector positions or joint angle vectors) has to be found, which models the special kinematic and dynamic properties of a robot. Therefore the controller f has to realise a mapping

between the 3D data of the end effector position and the joint angle configurations of the robot. Such a mapping lacks from accuracy due to manufacturing tolerances of the robot components and wear (like other mechanical systems).

It is essential to identify the individual physical robot parameters as well as the strategy the controller uses for determining trajectories in order to build accurate controller models. It is obvious that both procedures are learning processes, which have the potential to be implemented e.g. using Artificial Neural Networks.

2 Artificial Neural Networks

In the recent years various types of artificial neural networks (ANN) have been developed, studied and thereafter applied to practical and theoretical research fields related to the computational learning of natural systems like pattern and speech recognition, machine learning etc.
The success of ANN's may be explained by the fact, that they can be simply implemented either in software or hardware.

A neural network provides a method of mapping between high dimensional input/output spaces. The information held in the ANN (In this context knowledge about mapping joint angle- and Cartesian space) is stored in the form of weighted connections between artificial neurons, which are generally organised in layers [4]. Adjusting the values of the weights during what is known as the training process configures the ANN.

During the training process the network is fed with example pairs of joint angle- and respective Cartesian end effector vectors. In order to provide a proper mapping, the weights of the net have to be adjusted using an appropriate learning rule. Learning rules vary according to the type of network and learning strategy examples are:

i. The *Hebbian rule* is applied to unsupervised learning in for example single layer perceptrons;
ii. The *Backpropagation* rule used in *supervised learning* of multi-layer perceptrons.

After finishing the training process the ANN is be able to classify incoming patterns (joint angles vectors) and to provide the proper mapping. The ability of ANN to give reasonable output for inputs not contained in the training set is known as *generalisation*.
This property makes an ANN in general a useful approximation tool [12] e.g. for learning the behaviour and modelling of highly non-linear mechanisms [5][8] such as robot arms [13].

Nevertheless, neural models are pure numerical computational models. To convert the information contained in the weight matrices of the net into a readable symbolic form, in order to extract information about the learning state of the network, is a difficult task [11].
However, the key problem a designer of an ANN has, is in selecting an appropriate net structure for the mechanism to be learned. This means the right choice of the network type (either a *feed forward* or a *recurrent network*), the number of the

neurons, layers, suitable activation functions e.g. *hyperbolic tangent* or *sigmoidal* within the neurons, all have to be determined empirically [15]. Another problem is the initialisation of the weight matrices of the ANN. Tests have shown that non-optimal weight configurations lead to a considerable longer training time especially when applying backpropagation [10].
Furthermore during the training of the net with pairs of example patterns, which is an iterative optimisation process, the algorithm may run into a local minimum [9]. This means, that the weight values of the ANN have not been optimally adjusted to the new example. The example has then not been correctly learned by the ANN. In this context a point of concern is also the capacity of the ANN, which has to be analysed in order to avoid over-training errors [1].
Another well-known fact is, that ANN provide an undetermined mapping beyond the range they have been taught (locality). For the application it means, that a possible critical value that exceeds the admissible range of mapping has to be considered and the ANN to be taught with respective examples.

3 Analytical Models

The more traditional engineering approach is to build analytical models of the robot using system equations. The precondition therefore is the knowledge of the respective kinematic, dynamic and controller parameters. Whereas the kinematic (length and twist of links etc.) and dynamic (mass of links etc.) parameters are mostly supplied by the manufacturer of the robot, the controller algorithms are in general not documented. This makes it generally difficult to find a closed analytical description for the entire mechanism.

However, using the kinematic parameters (pref. in *Denavit- Hartenberg (DH)* notation) a direct forward kinematic model (coarse geometric model) can normally be generated. To work towards more accuracy the manufacturing tolerances of the physical parameters of the individual robot have to be integrated into the model as parameters that have to be identified experimentally. Therefore new models or refinements of the DH model have been proposed [14][16]. To cover non-linearity's like joint boundaries etc. respective constraints have also to be added to the model. The growing number of parameters raises model complexity and may impair the solvability of the inverse transformations[48] of the model to compute end effector positions into the joint angle vector space.

Nevertheless, a complete or partial analytical model (provided it exists) offers many advantages. The generation of the nominal kinematic models can be conveniently performed with the aid of computer algebra systems like *Maple* or *Mathematica* [17]. Using such tools the equations can be manipulated exclusively symbolically, which allows flexible modelling[49] and guarantees maximal stability and minimises the overhead of a subsequent numerical computation.

Several software toolboxes like *robotica* [7] provide predefined methods for calculating dynamics e.g. *Euler- Lagrange* equations of motion of the model.

[48] mostly iterative solution of non-linear equations
[49] system equations can be dynamically expanded with parameter terms

However, in general these packages are straight- forward implementations (based on matrix algebra) of the standard robot modelling concepts without really taking advantage of the possibilities of dynamic structuring and programming combined with symbolic representation and computation provided by the computer algebra system.

4 Conclusion

In this paper the advantages and disadvantages of analytical and artificial neural robot model are discussed. The essential features of both approaches may be summarised in the following table:

	Analytic Model	**Neural Model**
Limitations of modelling methodology	- many non-linearities - high complexity in general	- training examples must be provided - extrapolation
Effort involved in Model generation	- initially relative simple: generation of a nominal kinematic model with DH-parameters	- a suitable architecture (numbers of neurons and layers) has to be chosen empirically - high training effort
Task variability	- covers all modelled (described with the equations) situations	- models only trained tasks
Complexity	- mostly high: many kinds of model configurations like singularities have to be considered - complexity raises with complexity of the robot (non- linearities, Degrees of Freedom) and the number of parameters to be identified	- only connectionist comp-lexity (between neurons)
computer internal representation	- symbolic descriptive	- numerical
accuracy	- depends on accuracy of the model	- depends on number of training examples
learning capacity	- dynamic growing	- in general restricted, except by referring an architecture like in [2]

Table 1 : Summarise of analytic and neural models

In general analytic modelling of robot mechanisms is more difficult because of the necessity of a mathematical description of complex non-linearities. The quality of a neural model depends on the choice of the network architecture and the number and quality of trained examples.

Although artificial neural nets have been widely studied and successfully applied to numerous specialised research projects, until now they have been in more empirical disciplines.
But with better computing facilities and the aid of dynamic programming and computer algebra techniques it has become worthwhile to investigate in general analytical models rather then separated solutions of subtasks modelled by artificial neural nets.

5 References

[1] S. Amari, N. Murata, K.-R. Müller, M. Finke, H. Yang
Asymptotic Statistical Theory of Overtraining and Cross-Validation
METR 95-06 August 1995

[2] Fritzke, Bernd, *Growing Cell Structures - A Self-organizing Network for Unsupervised and Supervised Learning*. TR-93-026, May 1993, International Computer Science Institute Berkeley, California

[3] Hsinchun Chen. *Machine Learning for Information Retrieval: Neural Networks, Symbolic Learning, and Genetic Algorithms*
Journal of the American Society for Information Science, 1994, in press.

[4] Kröse, B.J.A. and Smagt, P.P. van der. *An Introduction to Neural Networks*.
University of Amsterdam, Amsterdam, The Netherlands, 1994.

[5] F. Lange and G. Hirzinger, *Learning to Improve the Path Accuracy of Position Controlled Robots*, IEEE/RSJ/GI Int. Conference on Intelligent Robots and Systems IROS'94, München, Germany, Sept. 1994

[6] Lippmann, R.P., *An introduction to computing with neural networks.* IEEE Acoustics Speech and Signal Processing Magazine, 4(2):4-22, April 1987.

[7] J. F. Nethery, *Robotica: A structured environment for computer aided design and analysis of robots,* Master's thesis, University of Illinois, Urbana, IL, 1993. Department of Electrical and Computer Engineering.

[8] Poggio, T. & Girosi, F. (1989), *A theory of networks for approximation and learning*, Technical Report AIM-1140, Artificial Intelligence Laboratory and Center for Biological Information Processing, Whitaker College, Massachusetts Institute of Technology.

[9] T. Poston, C. Lee, Y. Choie, and Y. Kwon. *Local minima and backpropagation,* in International Joint Conference on Neural Networks, vol.2, (Seattle, (WA)), pp. 173-176, IEEE Press, July 1991

[10] Shavlik Jude W., Raymond J. Mooney, Geoffrey G. Towell. *Symbolic and Neural Learning Algorithms: An Experimental Comparison.* TR 857, Computer Sciences Department, University of Wisconsin-Madison, June 1989.

[11] Shavlik Jude W. *A Framework for Combining Symbolic and Neural Learning*. TR 1123, Computer Sciences Department, University of Wisconsin-Madison, WI, November 1992.

[12] Smagt P. van der, and F. Groen. *Approximation with neural networks: Between local and global approximation*. In Proceedings of the 1995 International Conference on Neural Networks, pages II:1060-II:1064, 1995. (Invited paper).

[13] Smagt, P. van der and Schulten, K. *Control of pneumatic robot arm dynamics by a neural network*. In World congress on neural networks pg III/180--183. Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, Portland, OR, USA, 1993.

[14] Stone, Henry W. *Kinematic modelling, identification, and control of robotic manipulators. Dissertation.* Carnegie Mellon University, 1986

[15] Vysniauskas, V. and Groen, F.C.A. and Kröse, B.J.A. *The optimal number of learning samples and hidden units in function approximation with a feedforward network.* CS-93-15, 1993.

[16] Whitney, D. E. and Lozinski, C.A. *Industrial Robot Calibration Methods and Results*. In the Proceedings of the international Computers in Engineering Conference, pages 92-100. ASME, New York, August, 1984. Las Vegas, NV.

[17] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer.* New York, NY: Addison-Wesley Publishing Company, Inc, 1991.

Appendix C

C++ Source code

Symbol definitions

```
/************************************************************************
   toolunit.h
   Jens- Uwe Dolinsky
   07.03. 2001
************************************************************************/
#ifndef _TOOLUNIT
#define _TOOLUNIT


#define bool  unsigned short
#define true  1
#define false 0
#define max_token_length 200


typedef char symbolstringtyp[max_token_length];     // for Scanner


enum Tsymbole {BEGINNING,       ENDTOKEN,
               IDENTIFIER,
               DOT,
               INVALID,         OPEN_PARAN,          CLOSE_PARAN,
               DOUBLECROSS,
               FUNC_TOKEN,      VARIABLE,            STRING_CONST,
               NUM_TOKEN,       FLOATNUMBER,
               GP_DIVISION,
               MULTIP_TOKEN,    DIV_TOKEN,            ADD_TOKEN,
               SUB_TOKEN,
              };

#define dash_operator(c) ((c==SUB_TOKEN)||(c==ADD_TOKEN))
#ifdef MATRIX_GA
#define dot_operator(c)  ((c==MULTIP_TOKEN)||(c==DIV_TOKEN)||(c==GP_DIVISION))
#else
#define dot_operator(c)  ((c==MULTIP_TOKEN)||(c==DIV_TOKEN))
#endif

#endif
```

GP tree implementation

```
/************************************************************************
   knot_tab.h
************************************************************************/
#include <string.h>


#ifndef _KNOTTAB
#define _KNOTTAB
```

```
#ifdef NO_STRING_CLASS
#define STRING_TO_CHARPT get_local_string()
class String
{ private:
    int mem_size,length;
    char* ptr;
    void append(const char *);
    int my_strlen(const char*s)
     { if (s==NULL) return 0;
       return strlen(s);
     }
    void init();
 public:
    const char* get_local_string()
    { return ptr; }
    String(String &);
    String(const char*);
    String();

    String& operator+=(const char*);
    String& operator=(String&);
    ~String();
};


typedef String string_class;
#else
#ifdef COMPILE_FOR_LINUX
#include <string>
typedef string string_class;
#define STRING_TO_CHARPT c_str()

#else //Watcom
#include <string.hpp>
typedef String string_class;
#define STRING_TO_CHARPT operator char const*()
#endif
#endif



#include <stdio.h>
#include "toolunit.h"


class referenztyp //smart pointer management
{private:
  signed long    references;
 protected:
  virtual       ~referenztyp(){}
  referenztyp() { references = 0;}
 public:
   inline void create_reference()       {references++;}
   static void remove_one_reference(referenztyp*);
};


class Node;
typedef Node *PNode;


class klisttyp : public referenztyp
{ friend class node_managertype;
 protected:
  void set_pred(klisttyp *p) {Predecessor=p;}
  void set_succ(PNode s) {Successor=s;}
 public:
  klisttyp *linlist_predecessor;
  PNode   linlist_successor;
  klisttyp *Predecessor;
  PNode  Successor;
  inline klisttyp(klisttyp*);
  inline klisttyp();
  virtual ~klisttyp();
};



class Node_page: public klisttyp
{ public:
    int        breite,hoehe;
    PNode    lnext,rnext;
    PNode    aequivalent;
    double     double_value;
    int number_of_nodes;
    void           init_aequivalences();
    inline Node_page(PNode,PNode,klisttyp*);
    virtual ~Node_page();
};


class node_managertype: public referenztyp
{ private: klisttyp vars_,numbers_,stringconst_,floats_,
           funcs_,mult_,div_,add_,sub_,else_;
  public:
  klisttyp linear_list;
  klisttyp *linear_last_element;
```

```
  node_managertype()
  { linear_last_element=&linear_list;
  }
  public:
  virtual ~node_managertype(){}
  PNode searchNode(const char*,const Tsymbole,const PNode,const PNode);
  PNode getFunctionNode(const char*,const PNode);
};

#ifdef COMPILE_FOR_LINUX
#define STRING_COMPARE(a,b) strcasecmp(a,b)
#else
#define STRING_COMPARE(a,b) strcmpi(a,b)
#endif


class Node :public Node_page
{ friend class node_managertype;
  private:
    char             *constant;
    node_managertype *node_list;
    Node(const Tsymbole,Node*,Node*,node_managertype*,klisttyp*);
    Node(const char*,const Tsymbole,node_managertype*,klisttyp*);
    Node(const char*,PNode,node_managertype*,klisttyp*);
    virtual ~Node();

 public:
   static PNode createNode(const char*,const Tsymbole,const PNode,const
PNode);//constructs new nodes

   PNode getFunctionNode(const char* name,const PNode args)
   { return node_list->getFunctionNode(name,args);
   }
   PNode get_node(const PNode h)
   { return get_node(h->wert(),h->symbol,h->lnext,h->rnext);
   }
   PNode get_node(const char*     symbolstring,
                  const Tsymbole  scannersymbol,
                  const PNode   next_left,
                  const PNode   next_right)
   { return node_list->searchNode(symbolstring,scannersymbol,next_left,next_right);
   }

    int         compare_identifiers(const char* s)              {return
STRING_COMPARE(wert(),s);}
    static int  compare_identifiers(const char*s1,const char*s2){return
STRING_COMPARE(s1,s2);}
    const char  *get_Nodevalue();
    const       Tsymbole  symbol;
    const char   *wert() {return constant;}
   static PNode get_float_node(PNode);
   void output_infix__(string_class*,Tsymbole=ENDTOKEN,bool=false);
   void output_infix(FILE*);
   void calculate_list();
  };//Node



typedef class Node_class    //aquisition control
{ public:
  PNode n;
  Node_class(PNode h)
  { n=h;
    n->create_reference();
  }
  Node_class(char *s,Tsymbole sym)
  { n = Node::createNode(s,sym,NULL,NULL);
  }
  ~Node_class()
  { Node::remove_one_reference(n);
  }
} Node_wrapper;

#endif


//knot_tab.cpp
#include "knot_tab.h"

klisttyp::klisttyp(klisttyp *liste)
{  Predecessor=liste;
   { Successor=liste->Successor;
     if (liste->Successor != NULL)
         liste->Successor->Predecessor=this;
     liste->Successor=(PNode)this;
   }
}

klisttyp::klisttyp()
{ Predecessor=NULL;
  Successor=NULL;
  linlist_predecessor=NULL;
  linlist_successor=NULL;
}

klisttyp::~klisttyp()
{
```

```
   if (Predecessor!=NULL) Predecessor->set_succ(Successor);
   if (Successor!=NULL) Successor->set_pred(Predecessor);
}


Node_page::Node_page(PNode left,PNode right,klisttyp *liste)
  :klisttyp(liste)
{ lnext=left;
  rnext=right;
  aequivalent=NULL;
  number_of_nodes=1;
  if (lnext!=NULL)
    number_of_nodes+=lnext->number_of_nodes;
  if (rnext!=NULL)
    number_of_nodes+=rnext->number_of_nodes;
}

Node_page::~Node_page()
    {  remove_one_reference(lnext);
       remove_one_reference(rnext);
       remove_one_reference(aequivalent);
    }

void Node_page::init_aequivalences()
{
    if (lnext!=NULL) lnext->init_aequivalences();
    if (rnext!=NULL) rnext->init_aequivalences();
    remove_one_reference(aequivalent);
}



void referenztyp::remove_one_reference(referenztyp* h)
{ if (h!=NULL)
  {
    if (--h->references == 0)
      delete h;
  }
}



#include <stdlib.h>//fuer atof in matrix_ga mode

//Constructor for Leaf node
Node::Node(const char*       s,
               const Tsymbole   sym,
               node_managertype *node_liste,
               klisttyp         *last_element)
       :Node_page(NULL,NULL,last_element),symbol(sym)

{
  constant = new char[strlen(s)+1];
  strcpy(constant,s);

  node_list=node_liste;
  node_list->create_reference();
  if ((sym==NUM_TOKEN)||(sym==FLOATNUMBER)) double_value=atof(s);
  linlist_predecessor=NULL;
  linlist_successor=NULL;
}

//constructor for function node
Node::Node(const char*      s,
               PNode          argument,
               node_managertype *node_liste,
               klisttyp         *last_element)
       :Node_page(NULL,argument,last_element),symbol(FUNC_TOKEN)
{
  constant = new char[strlen(s)+1];
  strcpy(constant,s);

  node_list=node_liste;
  node_list->create_reference();
  linlist_predecessor=node_list->linear_last_element;
  linlist_predecessor->linlist_successor = this;
  linlist_successor=NULL;
  node_list->linear_last_element=this;
}//constructor

Node::Node(const Tsymbole sym,Node *links,Node *rechts,
               node_managertype *node_liste,klisttyp *last_element)
       :Node_page(links,rechts,last_element),symbol(sym)
{
  constant=NULL;
  node_list=node_liste;
  node_list->create_reference();
  linlist_predecessor=node_list->linear_last_element;
  linlist_predecessor->linlist_successor = this;
  linlist_successor=NULL;
  node_list->linear_last_element=this;
}

Node::~Node()
{
```

```
 if (constant!=NULL) delete[] constant;

  if (node_list->linear_last_element==this)
  {   if (linlist_predecessor==NULL)
      { printf("\nAbnormal termination");
        throw int(200);
      }
      node_list->linear_last_element=linlist_predecessor;
  }
  if (linlist_predecessor!=NULL)
        linlist_predecessor->linlist_successor = linlist_successor;
  if (linlist_successor!=NULL)
        linlist_successor->linlist_predecessor = linlist_predecessor;
 referenztyp::remove_one_reference(node_list);
}

/*********************************************************************/
#include <math.h>
#include <float.h> //for _fpreset

#include <signal.h>

#define MATH_NOT_EVALUABLE 9
#define ACCESS_VIOLATION   10

int matherr( struct _exception*/* er*/)
{
#ifdef DEBUG_VALUES
/*switch (er->type)
{ case DOMAIN:  printf("\nA domain error has occurred , such as sqrt(-1e0)"); break;
  case SING:    printf("\nA singularity will result, such as pow(0e0,-2)");break;
  case OVERFLOW:printf("\nAn overflow will result, such as pow(10e0,100)");break;
  case UNDERFLOW:printf("\nAn underflow will result, such as pow(10e0,-100)");break;
  case TLOSS:   printf("\nTotal loss of significance will result, such as
exp(1000)");break;
  case PLOSS: printf("\nPartial loss of significance will result, such as
sin(10e70)");break;
  default :printf("\nunknown math exception");
}*/
#endif
    _fpreset();
   throw int(MATH_NOT_EVALUABLE);
}


void matherr_(int)
{
#ifndef COMPILE_FOR_LINUX
    _fpreset();
#endif
   signal(SIGFPE,matherr_);//needs to be reinstalled, otherwise default handling is
used
   throw int(ACCESS_VIOLATION);
}

class error_treatment
{public:
  error_treatment()
  {
#ifndef COMPILE_FOR_LINUX
    _fpreset();
#endif
    signal(SIGFPE,matherr_);
 //   _set_matherr(&my_matherr);
  }
};
static error_treatment er;


/*
static int my_int()
{
  _set_matherr( &matherr );
  return 0;
}

static int iopu=my_int();
*/

#ifdef MATRIX_GA
void Node::calculate_list()
{int l=0;
 // PNode h;
 //_fpreset();
 /*try{*/
 for (PNode h=node_list->linear_list.linlist_successor;
       h!=NULL;
       h=h->linlist_successor,l++)
 {
  switch(h->symbol)
  { case ADD_TOKEN: h->double_value = h->lnext->double_value + h->rnext-
>double_value;continue;
    case SUB_TOKEN:h->double_value = h->lnext->double_value - h->rnext-
>double_value;continue;
    case MULTIP_TOKEN:h->double_value = h->lnext->double_value * h->rnext-
>double_value;continue;
    case GP_DIVISION: if (h->rnext->double_value == 0)
```

```
                      { h->double_value = 1;//closure condition
                        continue;
                       }//else perform a normal division
    case DIV_TOKEN:h->double_value = h->lnext->double_value / h->rnext-
>double_value;continue;
    case FUNC_TOKEN: if (h->compare_identifiers("SIN")==0)
                     { h->double_value = sin(h->rnext->double_value);
                       continue;
                     }
                     if (h->compare_identifiers("COS")==0)
                     { h->double_value = cos(h->rnext->double_value);
                       continue;
                     }
                     if (h->compare_identifiers("SQRTp")==0)
                     { if (h->rnext->double_value<0)
                          h->double_value = sqrt(-h->rnext->double_value);
                        else h->double_value = sqrt(h->rnext->double_value);
                       continue;
                     }
                     if (h->compare_identifiers("SIGN")==0)
                     { if (h->rnext->double_value>0)
                         h->double_value = 1;
                        else if (h->rnext->double_value<0)
                                 h->double_value = -1;
                               else h->double_value = 0;

                       continue;
                     }
                     if (h->compare_identifiers("LOG")==0)
                     { if (h->rnext->double_value==0)
                          h->double_value = 1;
                        else if (h->rnext->double_value<0)
                                 h->double_value = log(-h->rnext->double_value);
                               else h->double_value = log(h->rnext->double_value);
                       continue;
                     }

                     printf("\nFunction <%s> not implemented",h->wert());
    default:    printf("\nnot implemented math function");
                throw int(MATH_NOT_EVALUABLE);
  }
 }
}
#endif
```

```
PNode node_managertype::getFunctionNode(const char* name,const PNode args)
{  klisttyp *h=&funcs_;
   int k;
   while (h->Successor!=NULL)
   {   k=h->Successor->compare_identifiers(name);
       if (k==0) //descriptor must match
       { if (h->Successor->rnext > args)
            break;
           else if (h->Successor->rnext == args) //and the argument as well
                   return h->Successor;
       }
       if (k>0) //weitersuchen nicht mehr notwendig
          break;//while
       h=h->Successor;
   }
   args->create_reference();
   return new Node(name,args,this,h);
}


PNode node_managertype::searchNode(const char* s,const Tsymbole sym,const PNode
l,const PNode r)
{ klisttyp *h;
  int k;
  if (sym==VARIABLE)        h=&vars_;           else
  if (sym==NUM_TOKEN)           h=&numbers_;        else
  if (sym==FLOATNUMBER)       h=&floats_;         else
  if (sym==STRING_CONST) h=&stringconst_;    else
  if (sym==FUNC_TOKEN)
  {  return getFunctionNode(s,r);
  } else //binary operations
      { switch(sym)
        {case MULTIP_TOKEN: h=&mult_; break;
         case DIV_TOKEN: h=&div_;  break;
         case ADD_TOKEN:  h=&add_;  break;
         case SUB_TOKEN: h=&sub_;  break;
         default:    h=&else_; //all other operations
                     while (h->Successor!=NULL)
                     { if (h->Successor->symbol == sym)
                        if (h->Successor->lnext == l)
                         if (h->Successor->rnext == r)
                           return h->Successor;
                       h=h->Successor;
                     }
                     l->create_reference();
```

```
                    r->create_reference();
                    return new Node(sym,l,r,this,h);
        }
        while (h->Successor!=NULL)//+-/* operations
        { if (h->Successor->lnext > l)
             break; else
          if (h->Successor->lnext == l)
            if (h->Successor->rnext == r)
              return h->Successor;
          h=h->Successor;
        }
        r->create_reference();
        l->create_reference();
        return new Node(sym,l,r,this,h);
      }//else
  while (h->Successor!=NULL)
  { k=h->Successor->compare_identifiers(s);
    if (k==0) return h->Successor;
    if (k>0)
       return new Node(s,sym,this,h);
    h=h->Successor;
  }
  return new Node(s,sym,this,h);


}

PNode Node::createNode(const char* s,const Tsymbole sym,const PNode l,const PNode r)
//constructs new nodes
{  node_managertype* node_list_= new node_managertype();
   PNode h=node_list_->searchNode(s,sym,l,r);
   h->create_reference();
   return h;
}

const char *Node::get_Nodevalue()
{ switch (symbol)
  { case MULTIP_TOKEN          : return "*";
    case DIV_TOKEN           : return "/";
    case ADD_TOKEN            : return "+";
    case SUB_TOKEN           : return "-";
#ifdef MATRIX_GA
    case GP_DIVISION    : return "%";
#endif
    default    : return wert();
  }
}


void Node::output_infix(FILE *stream)
{ string_class s="";
  output_infix__(&s);
  fprintf(stream,
#ifdef GP_SYSTEM
  "%s"
#else
  "\n%s"
#endif
  ,s.STRING_TO_CHARPT);
}



void Node::output_infix__(string_class* str,Tsymbole op_vater,bool right_branch)
{ bool with_brackets=false;
  switch(symbol)
  { case NUM_TOKEN:
    case STRING_CONST:
          if ((compare_identifiers("0")==0)&&(dash_operator(op_vater)))
             return;
    case VARIABLE:  *str+=get_Nodevalue();return;
    case FUNC_TOKEN:  *str+=get_Nodevalue();
                    *str+="(";
                    rnext->output_infix__(str);
                    *str+=")";
                    return;
    case MULTIP_TOKEN:
               if ((lnext->symbol==GP_DIVISION)||(rnext->symbol==GP_DIVISION))
                  { with_brackets=true; break;}
    case GP_DIVISION:
    case DIV_TOKEN: if (op_vater==DIV_TOKEN) { with_brackets=true; break;}
               if (op_vater==GP_DIVISION) { with_brackets=true; break;}
              break;
    case SUB_TOKEN:
    case ADD_TOKEN:  if (dot_operator(op_vater)) with_brackets=true;
               if ((op_vater==SUB_TOKEN)&&(right_branch)) with_brackets=true;
  }
  if (with_brackets) *str+="(";
  if (lnext!=NULL) lnext->output_infix__(str,symbol);
  *str+=get_Nodevalue();
  if (rnext!=NULL) rnext->output_infix__(str,symbol,true);
  if (with_brackets) *str+=")";
}


#ifdef NO_STRING_CLASS
```

```
    void String::append(const char*s)
    { if (s==NULL) return;
      int  l=my_strlen(s);
      if (l!=0) //string contains something
      { char *ch_ptr;
        if ((length+l)>mem_size)
        { do mem_size+=1000;
          while ((length+l) > mem_size);
          ch_ptr=new char[mem_size];
          if (ptr==NULL)  //if current string empty
              ch_ptr[0]=0;
           else { strcpy(ch_ptr,ptr);//copy old string
                  delete []ptr;}     //delete old memory
          ptr=ch_ptr;
        }
        strcat(ptr,s);          //concatenate string
        length+=l;
      }
    }

    void String::init()
    { ptr=NULL;
      mem_size=length=0;
    }
//3 constructors
    String::String(const char*s)
    { init();
      append(s);
    }
    String::String(String &h)
    { init();
      append(h.get_local_string());
    }
    String::String()
    { init();
    }
//operators
    String& String::operator+=(const char* s)
    { append(s);
      return *this;
    }
    String& String::operator=(String& h)
    { if (ptr!=NULL)
         delete []ptr;
      init();
      append(h.get_local_string());
      return *this;
    }
    String::~String()
    { if (ptr!=NULL)
        delete[] ptr;
    }
#endif
```

GP type definitions

```
#ifndef GA_TYPES__
#define GA_TYPES__
#include "matrix.h"

enum  var_type {NONLINEAR,LINEAR};
typedef struct { char*    name;
                 double   value;
                 var_type type;
               } variable_type;


class parameter_generator_type
//generates a new parameter (string) for further processing
{  char  prefix[30];  //all parameter names starts with <prefix>
 public:
   int   parameter_number; //number of parameters generated
   char  actual_parameter[10];

  parameter_generator_type(char* name_prefix)
  { parameter_number=0;
    strcpy(prefix,name_prefix);
  }
  const char *generate_parameter()
  { sprintf(actual_parameter,"%s%i",prefix,++parameter_number);
    return actual_parameter;
  }
};

#include "my_templates.h"
#include <math.h>
class double_vector: public m_vector<double>
{public:
  double_vector(int d):m_vector<double>(d){}
  double sum_of_squares()
  { double z=0;
    for (int l=0; l<dimension; l++)
```

```
      z+=(data[l]*data[l]);
    return z;
  }
  double norm()
  { return sqrt(sum_of_squares());
  }
  void dump(char*);
};


class nodelist_type
{
  protected:
   PNode node_db;
  public :
   PNode *list;
   int     count;

   void list_add(const char*name,double value)
   { list[count]=node_db->get_node(name,VARIABLE,NULL,NULL);
     list[count]->create_reference();
     list[count]->double_value=value;
     count++;
   }
   void add(const double_vector& v)
   {
     for (int l=0; l<count; l++)
       list[l]->double_value += v.data[l];
   }
   void backup_in(double_vector &v)
   { for (int l=0;l<count; l++)
       v.data[l]=list[l]->double_value;
   }
   void assign(const double v[])
   { for (int l=0; l<count; l++)
       list[l]->double_value=v[l];
   }
   void assign(const double_vector &v)
   { for (int l=0; l<count; l++)
       list[l]->double_value=v.data[l];
   }
   void print(FILE *ostr)
   { for (int l=0; l<count; l++)
       fprintf(ostr,"\n%s   ,%lf",list[l]->wert(),list[l]->double_value);
     fflush(ostr);
   }
   nodelist_type(int size__,PNode node_base)
   { list=new PNode[size__];
     count=0;
     node_db=node_base;
   }
   ~nodelist_type()
   { while (count--)
       referenztyp::remove_one_reference(list[count]);
     delete []list;
   }
};

class parameterlist_type :public nodelist_type
{
 public:
  nodelist_type linear,nonlinear;
  parameterlist_type(int size__,PNode node_base):
      nodelist_type(size__,node_base),
      linear(size__,node_base),nonlinear(size__,node_base)
   {}
  void add_to_list(const char*name,var_type type,double value)
   { list_add(name,value);
     if (type==LINEAR)
        linear.list_add(name,value);
      else nonlinear.list_add(name,value);

   }
  void initialise(const variable_type vars[],int count_)
  { for (int l=0; l<count_; l++)
      add_to_list(vars[l].name,vars[l].type,vars[l].value);
  }
};


class chromosome_type
{
 public:
  static double random();
  static int random(int);
};// *p_chromosome_type;


enum F_status {EVALUATED,NOT_EVALUATED};

class abstract_chromosome
{
public:
  F_status status;
  double   fitness;
  abstract_chromosome()
```

```
  { status=NOT_EVALUATED;
  }
};


typedef struct dataset_type//data format for calibration
{ double             theta[6]; //set of joint angles
  double             p[6];    //position measured by robotrak
} *p_dataset_type;

#endif


#include "ga_types.h"
#include <stdlib.h>


double chromosome_type::random()
{ double z= rand();
  z/=RAND_MAX;
  return z;
}


int chromosome_type::random(int range)
{ double z=range*rand();
  z/=RAND_MAX;
  z+=0.5;
  return (int)z;
}
```

GP resources

```
#ifndef __GP_RESOURCE__
#define __GP_RESOURCE__
static const MAX_SET_ELEMENTS=50;
enum gp_symbols {ADDITION,
                 MULTIPLICATION,
                 DIVISION,
                 SUBTRACTION,
                 VARIABLE_,
                 EPHEMERAL,
                 FUNCTION};

#include "ga_types.h"
class gp_i_set //parameters for GP system
{ public:
 struct
 { gp_symbols symbol;
   char       *s;
 } elements[MAX_SET_ELEMENTS];
 int number;
 gp_i_set()
 { number=0;}
 void add(gp_symbols sym,char *str)
 { elements[number].symbol=sym;
   elements[number].s=str;
   number++;
 }
 int get_element_randomly(/*PNode db*/)
 {  return chromosome_type::random(number-1);
 }
};

class gp_resource
{  Node_wrapper node_db;
  public:
   PNode create_random_terminal();
   gp_i_set terminals,
            functions;
   //gp_resource(PNode node_):node_db(node_)
   //{}
   double get_value() {return node_db.n->double_value;}
   void set_theta_value_and_compute_models(double v) const
   { node_db.n->double_value = v;
     node_db.n->calculate_list();
   }
   gp_resource(): node_db("theta",VARIABLE)
   {}
   const PNode get_nodeset() {return node_db.n;}
   PNode     create_tree(int,const bool=false);
   static bool valid_tree(const PNode);
   static int  tree_depth(const PNode);
   static PNode create_floatconst(PNode,double);

};

class gp_robot_chromosome_functions
{
 public:
  static int     get_number_of_next_const_node(const PNode,const int);
 protected:
```

```
  static bool    crossover(const PNode,const PNode,PNode[]);
  static PNode swap(const PNode,int,PNode);
  static PNode get_node(const PNode,int);
  static PNode create_tree(gp_resource&,int,bool);

};


class joint_section : private gp_robot_chromosome_functions
{ gp_resource    *sys;
  PNode        tree;
  PNode replace_random_leaf(const PNode);

 public:
  joint_section(gp_resource &s,int max_depth,bool full_depth)
  { tree = create_tree(s,max_depth,full_depth);
    sys=&s;
  }
  joint_section(gp_resource &s)
  { sys = &s;
    const PNode h = s.get_nodeset();
    tree = h->get_node(NULL,SUB_TOKEN,h->get_node("theta",VARIABLE,NULL,NULL),
                                      h->get_node("theta",VARIABLE,NULL,NULL));
    tree->create_reference();
  }
  joint_section(joint_section &s)//copy constructor; only used by GP
  { tree = s.tree;
    sys  = s.sys;
    tree->create_reference();
  }
  ~joint_section()
  { Node::remove_one_reference(tree);
  }
  const gp_resource* get_resource() const {return sys;}
  const PNode get_tree() const {return tree;}
  double tree_value() const {return tree->double_value+sys->get_value();}
  bool crossover(joint_section*);
  bool mutation();
  joint_section* create_copy()
  { return new joint_section(*this);
  }
  void print(FILE *f)
  { tree->output_infix(f);
  }
  bool equals(const joint_section *h)
  { return (h->tree==tree);
  }
  void mutate_constant(int,double);
  int number_of_next_const_node(const int number) const
  {  return get_number_of_next_const_node(tree,number);
  }
};

#endif


#include "gp_resource.h"
#include "gp_parameter.h"


PNode gp_robot_chromosome_functions::create_tree(gp_resource &gp_sets,int
max_depth,bool full_depth)
{ PNode temp;
  for(;;)
  {  temp=gp_sets.create_tree(max_depth,full_depth);
     temp->create_reference();
     if (gp_resource::tree_depth(temp)>=MIN_TREE_DEPTH)
        if (gp_resource::valid_tree(temp))
           return temp;
     Node::remove_one_reference(temp);
  }
}


int gp_robot_chromosome_functions::get_number_of_next_const_node
                (const PNode tree,const int from_node)
{ for (int l=from_node; l<=tree->number_of_nodes; l++)
    if (get_node(tree,l)->symbol==FLOATNUMBER)//very inefficient
      return l;
  return 0;
}


PNode gp_robot_chromosome_functions::get_node(const PNode node,int node_number)
{ if (--node_number ==0)
      return node;
  if (node->lnext!=NULL)
  { if (node->lnext->number_of_nodes >= node_number)
       return get_node(node->lnext,node_number);
    node_number -=node->lnext->number_of_nodes;
  }
  return get_node(node->rnext,node_number);
}
```

```
PNode gp_robot_chromosome_functions::swap(const PNode node,int node_number,PNode
new_node)
{ if (--node_number ==0)
     return new_node;
  if (node->lnext!=NULL)
  { if (node->lnext->number_of_nodes >= node_number)
      return node->get_node(node->wert(),
                            node->symbol,
                            swap(node->lnext,node_number,new_node),
                            node->rnext);
    node_number -=node->lnext->number_of_nodes;
  }
    return node->get_node(node->wert(),
                          node->symbol,
                          node->lnext,
                          swap(node->rnext,node_number,new_node));
}

static inline bool leafnode(const PNode h)
{ return ((h->lnext==NULL) && (h->rnext==NULL));
}

bool gp_robot_chromosome_functions::crossover(const PNode el_1,const PNode el_2,PNode
new_nodes[])
{ int c1,c2;
  if (chromosome_type::random()< POINT_CROSSOVER_RATE)
    for(int l=0;l<INVALID_ATTEMPTS;l++)
    {  c2=chromosome_type::random(el_2->number_of_nodes-1)+1;
       c1=chromosome_type::random(el_1->number_of_nodes-1)+1;
       new_nodes[0]=swap(el_1,c1,get_node(el_2,c2)); new_nodes[0]-
>create_reference();
       new_nodes[1]=swap(el_2,c2,get_node(el_1,c1)); new_nodes[1]-
>create_reference();
       if
(gp_resource::valid_tree(new_nodes[0])&&gp_resource::valid_tree(new_nodes[1]))
          return true;
       Node::remove_one_reference(new_nodes[0]);
       Node::remove_one_reference(new_nodes[1]);
    }
   else //no point mutation permitted
    {PNode k1,k2;
     for(int l=0;l<INVALID_ATTEMPTS;l++)
     {  c2=chromosome_type::random(el_2->number_of_nodes-1)+1;
        c1=chromosome_type::random(el_1->number_of_nodes-1)+1;
        k1 = get_node(el_1,c1);
        k2 = get_node(el_2,c2);
        if (leafnode(k1) && (leafnode(k2))) continue;
        new_nodes[0]=swap(el_1,c1,k2); new_nodes[0]->create_reference();
        new_nodes[1]=swap(el_2,c2,k1); new_nodes[1]->create_reference();
        if
(gp_resource::valid_tree(new_nodes[0])&&gp_resource::valid_tree(new_nodes[1]))
          return true;
        Node::remove_one_reference(new_nodes[0]);
        Node::remove_one_reference(new_nodes[1]);
     }
    }
  fputchar('C');
  return false;
}


PNode gp_resource::create_floatconst(PNode node_db,double val)
{ char   buffer[30];
  sprintf(buffer,"%0.16G",val);
  PNode h = node_db->get_node(buffer,FLOATNUMBER,NULL,NULL);
  h->double_value = val;
  return h;
}


PNode gp_resource::create_random_terminal()
{ int l = terminals.get_element_randomly();
  switch(terminals.elements[l].symbol)
  { case  VARIABLE : return node_db.n-
>get_node(terminals.elements[l].s,VARIABLE,NULL,NULL);
    //case  NUMBER   : return node_db.n-
>get_node(terminals.elements[l].s,NUM_TOKEN,NULL,NULL);
    case  EPHEMERAL: return create_floatconst(node_db.n,chromosome_type::random());
  }
  throw int(0);
}

PNode gp_resource::create_tree(int depth,const bool full_depth_required)
{ if (--depth == 0)
    return create_random_terminal();
  if (!full_depth_required)
    if (chromosome_type::random(1)==1)
      return create_random_terminal();
  int l = functions.get_element_randomly();
  switch(functions.elements[l].symbol)
  { case ADDITION:      return node_db.n->get_node(NULL,ADD_TOKEN,
create_tree(depth,full_depth_required),create_tree(depth,full_depth_required));
    case MULTIPLICATION:return node_db.n-
>get_node(NULL,MULTIP_TOKEN,create_tree(depth,full_depth_required),create_tree(depth,
full_depth_required));
```

```
    case DIVISION:      return node_db.n-
>get_node(NULL,GP_DIVISION,create_tree(depth,full_depth_required),create_tree(depth,f
ull_depth_required));
    case SUBTRACTION:   return node_db.n-
>get_node(NULL,SUB_TOKEN,create_tree(depth,full_depth_required),create_tree(depth,ful
l_depth_required));
    case FUNCTION:      return node_db.n-
>get_node(functions.elements[l].s,FUNC_TOKEN,NULL,create_tree(depth,full_depth_requir
ed));
   }
  printf("Illegal element called from create_tree");
  throw int(0);
  //return NULL;
}


int gp_resource::tree_depth(const PNode h)
{ if (h==NULL) return 0;
  int l=tree_depth(h->lnext);
  int r=tree_depth(h->rnext);
  return 1+(r>l?r:l);
}

bool gp_resource::valid_tree(const PNode h)
{ int depth=tree_depth(h);
  if (depth > MAX_TREE_DEPTH)
    return false;
  if (depth < MIN_TREE_DEPTH)
    return false;
  if (h->symbol==FUNC_TOKEN)
    if (h->rnext->wert()!=NULL)
     if (strcmp(h->rnext->wert(),"SIGN")==0)
      return false;
  return true;
}


bool joint_section::crossover(joint_section *rc)
{ PNode new_trees[2];
  if (gp_robot_chromosome_functions::crossover(rc->tree,tree,new_trees)==false)
     return false;
  Node::remove_one_reference(tree); //deleting old trees
  Node::remove_one_reference(rc->tree);
  tree     = new_trees[0];
  rc->tree = new_trees[1];
  return true;
}

PNode joint_section::replace_random_leaf(const PNode h)
{ switch(h->symbol)
   { case VARIABLE:
     case FLOATNUMBER:
     case NUM_TOKEN:     { PNode h2;
                       for (int l=0;l<INVALID_ATTEMPTS;l++)
                       {  h2 = sys->create_random_terminal();
                          if (h2!=h)
                            return h2;
                       }
                       return sys->create_random_terminal();
                     }
     case FUNC_TOKEN:  return h->get_node(h-
>wert(),FUNC_TOKEN,NULL,replace_random_leaf(h->rnext));
   }
  if (chromosome_type::random(1)==0)
    return h->get_node(h->wert(),h->symbol,replace_random_leaf(h->lnext),h->rnext);
  return h->get_node(h->wert(),h->symbol,h->lnext,replace_random_leaf(h->rnext));
}

void joint_section::mutate_constant(int number,double step)
{
   PNode h = get_node(tree,number);
   double val = h->double_value + step;
   h = swap(tree,number,gp_resource::create_floatconst(tree,val));
   h->create_reference();
   Node::remove_one_reference(tree);
   tree = h;
}

bool joint_section::mutation()//point or subtree mutation (Macro mutation))
{ int co_point;
  PNode h;
  double mut=chromosome_type::random();
  if (mut < POINT_MUTATION_RATE)
  { for (int l=0;l<INVALID_ATTEMPTS;l++)
    {
       co_point=chromosome_type::random(tree->number_of_nodes-1)+1;
       h = swap(tree,co_point,replace_random_leaf(get_node(tree,co_point)));
       h->create_reference();
       if (sys->valid_tree(h))
       {  Node::remove_one_reference(tree);
          tree = h;
          //status=NOT_EVALUATED;
          return true;//successful mutation
       }
       Node::remove_one_reference(h);
    }
  }
```

```
    else if (mut < POINT_MUTATION_RATE+SHRINK_MUTATION_RATE)
    { co_point=chromosome_type::random(tree->number_of_nodes-1)+1;
      int depth = gp_resource::tree_depth(get_node(tree,co_point));
      if (depth==1)
        for (int l=0;l<INVALID_ATTEMPTS;l++)
        { h = swap(tree,co_point,sys->create_random_terminal());
          h->create_reference();
          if (h!=tree)
          { Node::remove_one_reference(tree);
            tree = h;
            return true;//successful mutation
          }
          Node::remove_one_reference(h);
        }
          else
           { h = swap(tree,co_point,sys->create_tree(chromosome_type::random(depth-
2)+1,false));
             h->create_reference();
             Node::remove_one_reference(tree);
             tree = h;
             return true;
           }
    }
    else  for (int l=0;l<INVALID_ATTEMPTS;l++) //ordinary crossover
          { co_point=chromosome_type::random(tree->number_of_nodes-1)+1;
            h = swap(tree,co_point,sys->create_tree(INITIAL_MAX_TREE_DEPTH1,false));
            h->create_reference();
            if (sys->valid_tree(h))
            {  Node::remove_one_reference(tree);
               tree = h;
               //status=NOT_EVALUATED;
               return true;7//successful mutation
            }
            Node::remove_one_reference(h);
          }
  fputchar('M');
  return false;
}
```

GP chromosome

```
#ifndef __GP_CHROMOSOME__
#define __GP_CHROMOSOME__

#include "gp_resource.h"
#include "individual.h"

enum S_STAT_TYPE {MARKED,MODIFIED,UNMODIFIED};

class univariate_searcher:public abstract_chromosome
{ int            const_number;
  double         rate;
  double         fitness_before_change;
  S_STAT_TYPE    s_status;
  virtual void   mutate_constant(int,double)=0;
  virtual int    number_of_next_const_node(int) const =0;
  void           modify_rate();
 protected:
  void init__()
  { status=NOT_EVALUATED;
    s_status=UNMODIFIED;
  }
 public:
  int  number_of_constants() const;
  void change(int,double);
  univariate_searcher()
  { init__();
    rate=0.0;
    modify_rate();
  }
  virtual ~univariate_searcher(){}
  void mark();
};


class gp_robot_chromosome: public univariate_searcher
                         // public abstract_chromosome
{       //instance properties
  dataset_type   data[35];
  joint_section  joint;
  const int      theta_index_;


  void   assign_theta(int l,int j)  { data[l].theta[j]=joint.tree_value();}

 public:
  void add_joint_error(const double,const long);
  gp_robot_chromosome(gp_resource &s,int max_depth,bool full_depth,int _ind):
        joint(s,max_depth,full_depth),
        theta_index_(_ind)
  {}
  gp_robot_chromosome(gp_resource &s,int _ind)://creates one nominal instance 1*theta
        joint(s),
        theta_index_(_ind)
```

```
  {}
  virtual ~gp_robot_chromosome(){}
  const joint_section * get_joint() const {return &joint;}
  const dataset_type* get_local_data() const {return data;}
  double evaluate(long,kinematic_type*);
  void assign_points(const dataset_type[],const long);
  void replicate_thetavalues(const dataset_type[],const long,const int);
  double evaluate_m(const dataset_type[],long,kinematic_type*);
  static void evaluate_population(gp_robot_chromosome*population[],int
population_size,
                                  const dataset_type ds[],const long samples,
                                  kinematic_type* ind);
  void   apply_corrections(dataset_type[],const int/*,const int*/) const;

  void   crossover(gp_robot_chromosome *);
  void   mutation();
  gp_robot_chromosome* create_copy()
  { return new gp_robot_chromosome(*this);
  }
  void print(FILE *f)
  { joint.print(f); }
  bool equals(const gp_robot_chromosome *);
private:
  virtual void mutate_constant(int number,double step)
  { joint.mutate_constant(number,step);
  }
public:
  virtual int number_of_next_const_node(int n) const
  { return joint.number_of_next_const_node(n);
  }
//next constant must be called

};
#endif


#include "gp_chromosome.h"
#include "gp_parameter.h"

#ifdef PARALLEL_MODELLING
#define FROM_VALUE    0
#define TARGET_VALUE  1
#define CURRENT_VALUE 2
#endif

double gp_robot_chromosome::evaluate(long                rf_count,
                                     kinematic_type     *ptr_kinematic_model)
{   status  = EVALUATED;
    try
    {
#ifdef PARALLEL_MODELLING
         fitness = 0.0;
         for (int l=0; l<rf_count; l++)
           fitness+= fabs(data[l].theta[TARGET_VALUE]-data[l].theta[CURRENT_VALUE]);
#else
         fitness = ptr_kinematic_model->compute_position_error(data,rf_count);
#endif
    }
    catch(...)
    { printf(" FP ERROR");
      fitness= 100000000;//penalty
    }
  putchar('#');fflush(stdout);
  return fitness;
}

void gp_robot_chromosome::add_joint_error(const double delta_theta,const long
samples)
{ for (int l=0; l<samples; l++)
    data[l].theta[theta_index_]+=delta_theta;
}


void gp_robot_chromosome::evaluate_population(gp_robot_chromosome *population[],
                                                        int
population_size,
                                                        const dataset_type   ds[],
                                                        const long           samples,
                                                        kinematic_type     *ind)
  { const gp_resource *gpr = population[0]->joint.get_resource();
#ifdef ID_ONE_FITS_ALL
    for (int l=0; l<samples; l++)
      for (int j=ID_FROM_JOINT-1; j<ID_UP_TO_JOINT; j++)
      { gpr->set_theta_value_and_compute_models(ds[l].theta[j]);

        for (int p=0; p<population_size; p++)
          population[p]->assign_theta(l,j);
      }
#else
    const int _t_index = population[0]->theta_index_;
    for (int l=0; l<samples; l++)
#ifdef PARALLEL_MODELLING
      { gpr->set_theta_value_and_compute_models(ds[l].theta[_t_index]);
        for (int p=0; p<population_size; p++)
          population[p]->assign_theta(l,CURRENT_VALUE);
      }
#else
```

```
      { gpr->set_theta_value_and_compute_models(ds[l].theta[_t_index]);
        for (int p=0; p<population_size; p++)
          population[p]->assign_theta(l,_t_index);
      }
#endif
#endif
    for (int p=0; p<population_size; p++)
       population[p]->evaluate(samples,ind);
  }

void  gp_robot_chromosome::apply_corrections(dataset_type       data_samples[],
                                             const int          samples/*,
                                             const int          __theta_index*/)
const
{ const gp_resource *gpr = joint.get_resource();

  for (int l=0; l<samples; l++)
  {  gpr->set_theta_value_and_compute_models(data_samples[l].theta[theta_index_]);//
theta.list[0]->double_value = data[l].theta[j];
     data_samples[l].theta[theta_index_] = joint.tree_value();//tree[0]-
>double_value;
  }
}


void gp_robot_chromosome::replicate_thetavalues(const dataset_type data_samples[],
                                                const long samples,
                                                const int theta_index)
{ for (int l=0; l<samples; l++)
    data[l].theta[theta_index] = data_samples[l].theta[theta_index];
}


void gp_robot_chromosome::crossover(gp_robot_chromosome *rc)
{
  if (joint.crossover(&rc->joint)==true)
  { rc->init__();
    init__();
  }
}

void gp_robot_chromosome::mutation()//point or subtree mutation (Macro mutation))
{
  if (joint.mutation()==true)
     init__();
}

void univariate_searcher::modify_rate()
{
  rate = (.5-chromosome_type::random())*.001;
}

int univariate_searcher::number_of_constants() const
{ int counter = 0;
  for (int l=number_of_next_const_node(1);l!=0;l=number_of_next_const_node(l+1))
      counter++;
  return counter;
}

void univariate_searcher::mark()
{ if (s_status==UNMODIFIED)
    {  s_status = MARKED;
       rate=0.0;
    }
}


void univariate_searcher::change(int number,double val)
{
  switch(s_status)
  { case MARKED:     fitness_before_change=fitness;
                     const_number = number;
                     modify_rate();
                     mutate_constant(const_number,rate);
                     s_status=MODIFIED;
                     return;
    case UNMODIFIED: fitness_before_change=fitness;
                     rate = val;
                     const_number = number;
                     mutate_constant(number,val);
                     s_status=MODIFIED;
                     return;
    case MODIFIED: if (fitness_before_change >= fitness)//carry on if
                   { if (number!=const_number)
                     { const_number = number;
                       modify_rate();
                     }
                     fitness_before_change=fitness;
                     mutate_constant(const_number,rate);
                     return;
                   }
                   modify_rate();
                   mutate_constant(const_number,rate);
  }
}
```

```
void gp_robot_chromosome::assign_points(const dataset_type ds[],const long samples)
{
#ifdef PARALLEL_MODELLING
  for (int l=0; l<samples; l++)
  {  data[l].theta[TARGET_VALUE] = ds[l+samples].theta[theta_index_];
     data[l].theta[FROM_VALUE]   = ds[l].theta[theta_index_];
  }
#else
  for (int l=0; l<samples; l++)
  { for (int p=0; p<3; p++)
      data[l].p[p] = ds[l].p[p];
    for (int j=0; j<6; j++)
      data[l].theta[j] = ds[l].theta[j];
  }
#endif
}

bool gp_robot_chromosome::equals(const gp_robot_chromosome *h)
{
   return joint.equals(&h->joint);
}
```

GP parameter

```
#ifndef __GP_PARAMETER
#define __GP_PARAMETER

#define INITIAL_MAX_TREE_DEPTH1 3 //initialisation of initial population :  beginning
of ramp
#define INITIAL_MAX_TREE_DEPTH2 6 //depth at end of ramp
#define MAX_TREE_DEPTH          9
#define MIN_TREE_DEPTH          2
#define INVALID_ATTEMPTS        19 //how many attempts to create a offspring before
giving up
                                   //for crossover and mutation
#define POPULATION_SIZE         200 //GA
#define NUMBER_GENERATIONS      30
#define TOURNAMENT_SIZE         5
#define CROSSOVER_RATE          0.7 //0.8
#define MUTATION_RATE           0.3 //0.2

#define POINT_CROSSOVER_RATE    0.2

#define POINT_MUTATION_RATE     0.3
#define SHRINK_MUTATION_RATE    0.2

#define ELITIST__
#define PARALLEL_MODELLING

/*****************************************
  End of parameter declaration

*****************************************/


#ifdef ID_ONE_FITS_ALL
#define WITH_FITTING
#define ID_FROM_JOINT 1
#define ID_UP_TO_JOINT LAST_JOINT
#endif

#endif
```

GP system and calibration system

```
#ifndef __GP_SYSTEM
#define __GP_SYSTEM

#include "dataset.h"
#include "gp_chromosome.h"
#include "gp_parameter.h"

class abstract_gp_system
{
  public:
 virtual void correct(dataset_type[],const long)=0;
 virtual void ga(const dataset_type[],const long,FILE*)=0;
};


class gp_system:public abstract_gp_system
{ gp_resource         gp_set;
  kinematic_type     &kinematic_model;
  double              nominal_fitness;
  gp_robot_chromosome *population[POPULATION_SIZE];
  gp_robot_chromosome *new_population[POPULATION_SIZE];
  const int           theta_index;
#ifdef WITH_FITTING
  PNode             joint_models[6];
#endif
```

```
  void           init(int,int,int,bool);
  void           init_half_and_half();
  bool           already_in_population(const gp_robot_chromosome*,const int);
  void           copy_populations(double);
  void           breed_population(double);
  void           apply_mutation_to_elements_having_fitness(double);
  void           mutate_population();
  void           reinitialise();
 public:
  int            get_index_of_fittest();
  int            breed_until_Improvement(const dataset_type[],const
long,FILE*,int&,int);
  void           replicate_thetavalues(const int,const long,const int);
  void           replicate_thetavalues(const dataset_type[],const long,const int);
 public:

  void print(FILE*);
  gp_system(const dataset_type[],const long,kinematic_type&,int);
  ~gp_system();

 virtual  void ga(const dataset_type[],const long,FILE*); //main function
  const gp_robot_chromosome* get_individual(int l) const {return population[l];}
 virtual  void correct(dataset_type[],const long);
 void write_statistic(FILE *f);
};

class calibration_system :public abstract_gp_system
{ gp_system                        gp1,gp2,gp3,gp4,gp5;
#if LAST_JOINT==6
  gp_system                        gp6;
#endif
  gp_system                        *gp_systems[LAST_JOINT];
  int                              best_index[LAST_JOINT];
  Node_wrapper                     node_db;
  kinematic_type_with_derivative  der;
  //double                           fitness;
 public:
  calibration_system(const dataset_type[],const long,kinematic_type&);
  virtual void ga(const dataset_type[],const long,FILE*); //main function
  virtual void correct(dataset_type[],const long);
  void write_statistic(FILE *f);
};
#endif


#include "gp_system.h"

calibration_system::calibration_system(const dataset_type ds[],const long
n,kinematic_type& i_)
    :gp1(ds,n,i_,0),
     gp2(ds,n,i_,1),
     gp3(ds,n,i_,2),
     gp4(ds,n,i_,3),
     gp5(ds,n,i_,4),
#if LAST_JOINT==6
     gp6(ds,n,i_,5),
#endif
     node_db("theta",VARIABLE),
     der(node_db.n,PUMA_parametric,6,tool)
  { gp_systems[0]=&gp1;
    gp_systems[1]=&gp2;
    gp_systems[2]=&gp3;
    gp_systems[3]=&gp4;
    gp_systems[4]=&gp5;
#if LAST_JOINT==6
    gp_systems[5]=&gp6;
#endif
    for (int l=0; l<LAST_JOINT; l++)
      best_index[l]=-1;
  }


#ifdef PARALLEL_MODELLING
 #define SEQUENTIAL_
#endif


void calibration_system::ga(const dataset_type poses[],const long samples,FILE*
logfile)
{
#ifdef SEQUENTIAL_
  int joint = 0;
#else
  int joint = der.select_joint(poses,samples,logfile);
#endif
  int number_of_generations=NUMBER_GENERATIONS;
  int i=-1;
  int hjoint;
  for (int g=0;; g++)
  {
    if (g>=number_of_generations)
    {
#ifdef SEQUENTIAL_
      if ((joint+1)<LAST_JOINT)
      { joint++;
        g=0;
        continue;
```

```
      }
#endif
      int gh = kinematic_type::get_integer_from_stdin("\nHow many generations: ");
      if (gh>0) number_of_generations+=gh;
         else break;
#ifdef SEQUENTIAL_
      hjoint = kinematic_type::get_integer_from_stdin("\nWhich joint 0-5: ");
      if (hjoint!=joint)
      { i = gp_systems[joint]->get_index_of_fittest(); //i is the index of the best
individual
#ifndef PARALLEL_MODELLING
        if (i!=-1)
        {  const dataset_type *ds=gp_systems[joint]->get_individual(i)-
>get_local_data();
           for (int p=0; p<LAST_JOINT; p++)
             if (p==joint) gp_systems[p]->replicate_thetavalues(i,samples,joint);
               else gp_systems[p]->replicate_thetavalues(ds,samples,joint);
        }
#endif
        joint=hjoint;
        printf("\nSwitch to joint %i",joint);
      }
    }
    i = gp_systems[joint]-
>breed_until_improvement(poses,samples,logfile,g,number_of_generations);
    if (i != -1) best_index[joint] = i; //i is the index of the best individual
#else
    }
    i = gp_systems[joint]-
>breed_until_improvement(poses,samples,logfile,g,number_of_generations);
    if (i != -1)
    { best_index[joint] = i; //i is the index of the best individual
      const dataset_type *ds=gp_systems[joint]->get_individual(i)->get_local_data();
      hjoint = der.select_joint(ds,samples,logfile);
       // hjoint = kinematic_type::get_integer_from_stdin("\nWhich joint 0-5: ");

      if (hjoint!=joint)
      { for (int p=0; p<LAST_JOINT; p++)
         if (p==joint) gp_systems[p]->replicate_thetavalues(i,samples,joint);
           else gp_systems[p]->replicate_thetavalues(ds,samples,joint);
        joint=hjoint;
        //fprintf(logfile,"\nSwitch to joint %i",joint);
        printf("\nSwitch to joint %i",joint);
      }
    }
#endif
  }
} //main function


int gp_system::get_index_of_fittest()
{ int index=0;
  for (int l=1; l<POPULATION_SIZE; l++)
    if (population[l]->fitness < population[index]->fitness)
      index = l;
  return index;
}


void gp_system::correct(dataset_type ds[],const long samples) //one model fits all
{
  const gp_robot_chromosome *best =population[get_index_of_fittest()];
 // for (int j=0; j<LAST_JOINT; j++)
  best->apply_corrections(ds,samples/*,j*/);
 // double fitness=kinematic_model.compute_position_error(ds,samples);
}

void calibration_system::correct(dataset_type ds[],const long samples)//individual
models
{
  for (int j=0; j<LAST_JOINT; j++)
    if (best_index[j]!=-1)
      gp_systems[j]->correct(ds,samples);
}


gp_system::gp_system(const dataset_type teachpoints[],
                     const long REF_SAMPLES,kinematic_type& m,
                     int _index )
          :kinematic_model(m)
          ,theta_index(_index)
{
    gp_set.terminals.add(EPHEMERAL,NULL);
    gp_set.terminals.add(VARIABLE_,"theta");

    gp_set.functions.add(ADDITION,NULL);
    gp_set.functions.add(SUBTRACTION,NULL);
    gp_set.functions.add(MULTIPLICATION,NULL);
    gp_set.functions.add(DIVISION,NULL);

    gp_set.functions.add(FUNCTION,"SIN");
    gp_set.functions.add(FUNCTION,"COS");
    gp_set.functions.add(FUNCTION,"SIGN");
    gp_set.functions.add(FUNCTION,"SQRTp");


    init_half_and_half();
```

```
    for (int p=0; (p<POPULATION_SIZE); p++)
      population[p]->assign_points(teachpoints,REF_SAMPLES);
    gp_robot_chromosome::evaluate_population(population,POPULATION_SIZE,
                                    teachpoints,REF_SAMPLES,&kinematic_model);
    nominal_fitness =
kinematic_model.compute_position_error(teachpoints,REF_SAMPLES);
    printf("\nnominal fitness: %0.16G, best fitness %0.16G"
                ,nominal_fitness,population[get_index_of_fittest()]->fitness);

}

gp_system::~gp_system()
  { for (int l=0; l<POPULATION_SIZE; l++)
      delete population[l];
  }


void gp_system::print(FILE*f)
  { for (int l=0; l<POPULATION_SIZE; l++)
     population[l]->print(f);
  }

void gp_system::replicate_thetavalues(const dataset_type ds[],
                                      const long samples,
                                      const int theta_index)
{ for (int l=0; l<POPULATION_SIZE; l++)
    population[l]->replicate_thetavalues(ds,samples,theta_index);
}


void gp_system::replicate_thetavalues(const int index,
                                      const long samples,
                                      const int theta_index)
{ const dataset_type *ds = population[index]->get_local_data();
  for (int l=0; l<POPULATION_SIZE; l++)
    if (l!=index)
      population[l]->replicate_thetavalues(ds,samples,theta_index);
}


bool gp_system::already_in_population(const gp_robot_chromosome *h,const int index)
{
  for (int l=0; l<index; l++)
    if (population[l]->equals(h))
      return true;
  return false;
}


void gp_system::init(int i1,int i2,int max_depth,bool full_depth)
{ gp_robot_chromosome *temp;
  for (int l=i1; l<i2; l++) //creating individuals in the intervall i1-i2
  {  for(;;)
     {  temp=new gp_robot_chromosome(gp_set,max_depth,full_depth,theta_index);
        if (already_in_population(temp,l))
        { delete temp;
          continue;
        }
        break;
     }
     population[l]= temp;
  }
}

#ifndef ID_ONE_FITS_ALL
#define WITH_ONE_NONRANDOM
#endif

void gp_system::init_half_and_half()
{ int interval_size = (POPULATION_SIZE
#ifdef WITH_ONE_NONRANDOM
  -1
#endif
    )/(INITIAL_MAX_TREE_DEPTH2-INITIAL_MAX_TREE_DEPTH1+1);
  int interval_half = interval_size/2;
  //for (int li=0
  int interval_start=
#ifdef WITH_ONE_NONRANDOM
  1;
  population[0]=new gp_robot_chromosome(gp_set,theta_index);
#else
  0;
#endif
  int interval_middle,interval_end;
  for (int depth=INITIAL_MAX_TREE_DEPTH1; depth<=INITIAL_MAX_TREE_DEPTH2; depth++)
  { interval_middle = interval_start + interval_half;
    interval_end    = interval_start + interval_size;

    init(interval_start ,interval_middle,depth,true);//full sized trees
    init(interval_middle,interval_end   ,depth,false);//arbitrarily sized trees
    interval_start +=interval_size;
  }
  init(interval_start,POPULATION_SIZE,INITIAL_MAX_TREE_DEPTH2,true);
}

void gp_system::copy_populations(double fittest)
```

```
{   int counter=0;
    for (int l=0;l<POPULATION_SIZE; l++)
    { delete population[l];
      population[l]=new_population[l];
      if (population[l]->fitness == fittest)
        if (counter++ > 0)
          population[l]->mutation();
    }
}


void tournament_pick(gp_robot_chromosome   *population[],
                     gp_robot_chromosome   *tournament[],
                     const int          tournament_size,
                     const int          population_size)
{ long number;
  for (int l=0; l<tournament_size; l++)
  { number=chromosome_type::random(population_size-1);
    for (int k=0; k<l; k++) //picking different individuals
      if (tournament[k]==population[number])
      {  number=chromosome_type::random(population_size-1);
         k=-1;
      }
    tournament[l] = population[number];
  }
}

static gp_robot_chromosome
        *get_fittest(gp_robot_chromosome *tournament[],const int tournament_size)
{ int index =0;
  for (int l=1; l<tournament_size; l++)
    if (tournament[l]->fitness < tournament[index]->fitness)
       index = l;
  return tournament[index];
}



#include <float.h>
//static FILE *logfile;

double print_fittest(FILE *stream,gp_robot_chromosome *population[],
                     int  population_size,
                     bool show_individual,
                     const int theta_index)
{ double z=population[0]->fitness;
  double average=z;
  int index=0;
  _fpreset();
  for (int l=1; l<population_size; l++)
  {if (population[l]->fitness < z)
   {  z=population[l]->fitness;
      index=l;
   }
   average+=population[l]->fitness;
  }
  average/=population_size;
  fprintf(stdout,"\njoint %i Average: %0.16G  Best_performance: %0.16G
",theta_index,average,z);
  fprintf(stream,"\njoint %i Average: %0.16G  Best_performance: %0.16G
",theta_index,average,z);
  if (show_individual==true)
      population[index]->print(stream);
  population[index]->print(stdout);

  fflush(stdout);
  return z;
}

void calibration_system::write_statistic(FILE *f)
{
  for (int l=0; l<LAST_JOINT; l++)
  { fprintf(f,"\nJoint %i\n",l+1);
    gp_systems[l]->write_statistic(f);
  }
}

void gp_system::write_statistic(FILE *f)
{
  print_fittest(f,population,POPULATION_SIZE,true,theta_index);
}

int get_index_of_worst(gp_robot_chromosome *population[],int population_size)
{  int     worst_index=0;
   for (int l=1;l<population_size;l++) //searching worst individual; function to be
tuned
       if (population[l]->fitness > population[worst_index]->fitness)
         worst_index=l;
   return worst_index;
}

gp_robot_chromosome* create_copy_of_fittest(gp_robot_chromosome *population[],int
length)
{ gp_robot_chromosome *fittest=population[0];
  for (int l=1; l<length; l++)
    if (population[l]->fitness < fittest->fitness)
      fittest = population[l];
```

```
  return fittest->create_copy();
}


int evaluate_pick(          gp_robot_chromosome *parent1,//best individual from
tournament 1
                            gp_robot_chromosome *parent2,//best individual from
tournament 1
                            //dataset_type         training[], long tr_count,
                          //  const dataset_type   reference[],long rf_count,
                            //PNode              knode_db,
                            gp_robot_chromosome *new_creations[])
{
   gp_robot_chromosome *p1= parent1->create_copy();
   gp_robot_chromosome *p2= parent2->create_copy(); //create local copies
                                                              // of both parents
   new_creations[0]=p1;
   new_creations[1]=p2;
   double rate_ = chromosome_type::random();
   if (rate_< CROSSOVER_RATE)
   {   p1->crossover(p2);
       return 2;
   }
   else if (rate_ < CROSSOVER_RATE+MUTATION_RATE)
        {  p1->mutation();
           delete p2;
           return 1;
        }
   delete p2;
   return 1;
}


int comp_const_mutations(const gp_robot_chromosome *fittest,gp_robot_chromosome
*new_population[])
{ int cnum = fittest->number_of_constants();
  if (cnum < (POPULATION_SIZE-10))
  { int cind = new_population[0]->number_of_next_const_node(1);
    for (int l=1; l<=cnum; l++)
    {  new_population[l]=new_population[0]->create_copy();
       new_population[l]->change(cind,0.001);
       cind = new_population[0]->number_of_next_const_node(cind+1);
    }
    return cnum;
  }
  return 0;
}

void gp_system::breed_population(double fittest)
{ gp_robot_chromosome *tournament1[TOURNAMENT_SIZE];
  gp_robot_chromosome *tournament2[TOURNAMENT_SIZE];
  gp_robot_chromosome *new_creations[2];
  int creations;
#ifdef ELITIST
    new_population[0]=create_copy_of_fittest(population,POPULATION_SIZE);
    int cnum = comp_const_mutations(new_population[0],new_population);
    new_population[0]->mark();
    for (int p=1+cnum; (p<(POPULATION_SIZE)); p++)
#else
    for (int p=0; (p<(POPULATION_SIZE)); p++)
#endif
    {
tournament_pick(population,tournament1,TOURNAMENT_SIZE,POPULATION_SIZE);//selection

tournament_pick(population,tournament2,TOURNAMENT_SIZE,POPULATION_SIZE);//selection
       creations = evaluate_pick(get_fittest(tournament1,TOURNAMENT_SIZE),
                                 get_fittest(tournament2,TOURNAMENT_SIZE),
                                       new_creations);
       new_population[p]=new_creations[0];
       if (creations==2)
       {  if (++p < (POPULATION_SIZE))
             new_population[p]=new_creations[1];
          else delete new_creations[1];
       }
    }
    copy_populations(fittest);
}

void gp_system::mutate_population()
{ printf("mutate population");
  int i=get_index_of_fittest();
  for (int l=0; l<POPULATION_SIZE; l++)
    if (l!=i)
      if (chromosome_type::random(1)==0)
        population[l]->mutation();

}

void gp_system::reinitialise()
{ printf("reinit population");
  gp_robot_chromosome *best=create_copy_of_fittest(population,POPULATION_SIZE);
  for (int l=0; l<POPULATION_SIZE; l++)
    delete population[l];
  init_half_and_half();
  delete population[0];
  population[0] = best;
```

```
}

int gp_system::breed_until_improvement(const dataset_type teachpoints[],
                                       const long         REF_SAMPLES,
                                       FILE               *log_file_,
                                       int&               generation,
                                       int                number_of_generations)
{ double fittest=population[get_index_of_fittest()]->fitness;
  double tempf;
  gp_robot_chromosome::evaluate_population(population,POPULATION_SIZE,
                                  teachpoints,REF_SAMPLES,&kinematic_model);

  for (int counter=0; generation<number_of_generations; generation++)
  {  printf("\ngeneration %i",generation);fflush(stdout);
     breed_population(fittest);
     gp_robot_chromosome::evaluate_population(population,POPULATION_SIZE,
                                  teachpoints,REF_SAMPLES,&kinematic_model);
     tempf=print_fittest(log_file_,population,POPULATION_SIZE,false,theta_index);
     if (fittest>tempf)
     { int index = get_index_of_fittest();
       population[index]->print(log_file_);
       return index;
     }
     if (++counter == 40)  //if no better individuals occurred within 40 generations
     { //mutate_population();
       reinitialise();  //reinitialise the population to introduce new genetic
material
       counter=0;
       gp_robot_chromosome::evaluate_population(population,POPULATION_SIZE,
                                  teachpoints,REF_SAMPLES,&kinematic_model);
       tempf=print_fittest(log_file_,population,POPULATION_SIZE,false,theta_index);
     }
  }
  return -1;
}


void gp_system::ga(const dataset_type teachpoints[],const long REF_SAMPLES,
                                    FILE *log_file_/*,const double minimum_fitness*/)
{ double fittest=10000,tempf;
  int number_of_generations=NUMBER_GENERATIONS;
  printf("\nNumber of generations: %u\nPopulationsize: %u\nTournamentsize:
%u\n",NUMBER_GENERATIONS,POPULATION_SIZE,TOURNAMENT_SIZE);
#ifdef ELITIST__
  printf("Elitist mode\n");
#endif
  for (int g=0;; g++)
  {
    if (g==number_of_generations)
    { int gh = kinematic_type::get_integer_from_stdin("\nHow many generations: ");
      if (gh>0) number_of_generations+=gh;
         else break;
    }
    printf("\ngeneration %i",g);fflush(stdout);
    breed_population(fittest);
    gp_robot_chromosome::evaluate_population(population,POPULATION_SIZE,
                                  teachpoints,REF_SAMPLES,&kinematic_model);
    tempf=print_fittest(log_file_,population,POPULATION_SIZE,false,theta_index);
    if (fittest!=tempf)
      fittest=tempf;
  }

  tempf=print_fittest(log_file_,population,POPULATION_SIZE,true,theta_index);
}
```

Homogenous Node matrix used by kinematic model

```
#ifndef MATRIX__
#define MATRIX__
#include "parser.h"

typedef char    *string_matrix_type[3][4];
typedef PNode node_matrix_type[3][4];
typedef double  double_matrix_struct[4][4];
typedef double  double_hmatrix[3][4];

/*
                          xx, yx, zx, px;
                          xy, yy, zy, py;
                          xz, yz, zz, pz;
                          0   0   0   1
  PNode xx, yx, zx, px;
  PNode xy, yy, zy, py;
  PNode xz, yz, zz, pz;
          0   0   0   1
*/
typedef class parameter_type
{public:
  char* name;
  Tsymbole  type;
// parameter_type(){}
 parameter_type(char *n,const Tsymbole t)
```

```
 { name=n;
   type=t;
 }
} *p_parameter_type;

class plain_nodelist_type
{ public :
   PNode *list;
   int     count;

   void list_add_element(PNode new_node)
   { list[count]= new_node;
     list[count]->create_reference();
     count++;
   }
   plain_nodelist_type(int size__)
   { list=new PNode[size__];
     count=0;
   }
   ~plain_nodelist_type()
   { while (count--)
       referenztyp::remove_one_reference(list[count]);
     delete []list;
   }
};

class h_matrix //homogenous matrix
{
 private:
  PNode zero,one;
  void initialise(node_matrix_type,const string_matrix_type);
  static void create_references(node_matrix_type);
  static void delete_matrix(node_matrix_type);
  PNode mult(PNode,PNode);
  PNode add(PNode,PNode);
  void translate(p_parameter_type,const int);
  void multiplication(node_matrix_type,node_matrix_type,bool=false);
  void r_multiplication(h_matrix&);
  void multiplication(node_matrix_type,const node_matrix_type,const
node_matrix_type,bool=false);
  void translate_x(p_parameter_type);
  void translate_y(p_parameter_type);
  void translate_z(p_parameter_type);
  void delete_rotation_matrix();
 public://for gp
  node_matrix_type matrix;
 public:
  void _add_tool(const double[]);
  const PNode node_set;
  h_matrix(const string_matrix_type,PNode);
  ~h_matrix();
  void r_multiplication(const string_matrix_type,bool=false);
};

extern const string_matrix_type ROTX,ROTY,ROTZ,IDENTITY_M;

#endif


#include "matrix.h"

void h_matrix::_add_tool(const double tool_[])
{
  char buffer[20];
  for (int l=0; l<3; l++)
    if (tool_[l]!=0.0)
     { parameter_type p(buffer,FLOATNUMBER);
       sprintf(buffer,"%6.6f",tool_[l]);
       switch (l)
       { case 0: translate_x(&p); break;
         case 1: translate_y(&p); break;
         case 2: translate_z(&p); break;
       }
     }
}


void h_matrix::translate(p_parameter_type p,const int pos_component)
{ PNode parameter=node_set->get_node(p->name,p->type,NULL,NULL);
//  transl_derivatives.list_add_element(parameter);    //add parameter name
  PNode h;
  for (int l=0; l<3; l++)
  { h=mult(matrix[l][pos_component],parameter);
    h=add(matrix[l][3],h);
    h->create_reference();
    referenztyp::remove_one_reference(matrix[l][3]);
    matrix[l][3]=h;
  }
}

void h_matrix::translate_x(p_parameter_type p)
{ translate(p,0);
}

void h_matrix::translate_y(p_parameter_type p)
{ translate(p,1);
}
```

```
void h_matrix::translate_z(p_parameter_type p)
{ translate(p,2);
}


void h_matrix::delete_matrix(node_matrix_type m)
{ for (int row=0; row<3; row++)
    for (int column=0; column<4; column++)
      if (m[row][column]!=NULL)
        referenztyp::remove_one_reference(m[row][column]);
}

void h_matrix::delete_rotation_matrix()
{ for (int row=0; row<3; row++)
    for (int column=0; column<3; column++)
      if (matrix[row][column]!=NULL)
      {
          referenztyp::remove_one_reference(matrix[row][column]);
          matrix[row][column]=NULL;
      }
}

void h_matrix::create_references(node_matrix_type m)
{ for (int row=0; row<3; row++)
    for (int column=0; column<4; column++)
      m[row][column]->create_reference();
}
//************************************
h_matrix::h_matrix(const string_matrix_type strings,PNode _node_set)
                  :node_set(_node_set)
{ //node_set=_node_set; //set of nodes making the matrix up
  zero=node_set->get_node("0",NUM_TOKEN,NULL,NULL);
  zero->create_reference();
  one=node_set->get_node("1",NUM_TOKEN,NULL,NULL);
  one->create_reference();
  initialise(matrix,strings);//nodes are already referenced
}

h_matrix::~h_matrix()
{ referenztyp::remove_one_reference(zero);
  referenztyp::remove_one_reference(one);
  delete_matrix(matrix);
}

void h_matrix::initialise(node_matrix_type         m,
                          const string_matrix_type strings)
{ Parsertyp parser;
  for (int row=0; row<3; row++)
    for (int column=0; column<4; column++)
      m[row][column]=
          parser.read_expression(strings[row][column],node_set);
                //only one variable expected
}
//************************************************



inline PNode h_matrix::mult(PNode left,PNode right)
{ if ((left==zero)||(right==zero)) return zero;
  if (left==one) return right;
  if (right==one) return left;
  return one->get_node(NULL,MULTIP_TOKEN,left,right);
}

inline PNode h_matrix::add(PNode left,PNode right)
{ if (left==zero)  return right;
  if (right==zero) return left;
  return one->get_node(NULL,ADD_TOKEN,left,right);
}


void h_matrix::multiplication(node_matrix_type result,
                              const node_matrix_type left,
                              const node_matrix_type right,
                              bool  zero_scaling)
//main function performes mult. of homogenous matricies
{ PNode set=left[1][1],h; //node set vom anderen Node
  for (int r=0; r<3; r++)
  { for (int c=0; c<4; c++)
    { //element with first product
      result[r][c]=mult(left[r][0],right[0][c]);
      for (int l=1;l<3;l++)//adding all products
      { h=mult(left[r][l],right[l][c]);
        if (h==zero) continue;
        result[r][c]=add(result[r][c],h);
      }
    }
    if (!zero_scaling) //m[3][3]!=0
      if (left[r][3]!=zero)
        result[r][3]=add(result[r][3],left[r][3]);
      //adding last element to column vector
  }
}
```

```
void h_matrix::multiplication(node_matrix_type m1,node_matrix_type m2,bool
zero_scaling)
{ node_matrix_type res;
  multiplication(res,m1,m2,zero_scaling);    //result <res> without references
  create_references(res);
  delete_matrix(matrix);  //delete old matrix
  memcpy(matrix,res,sizeof(res)); //copy new matrix
}

void h_matrix::r_multiplication(h_matrix &ns)
{ multiplication(matrix,ns.matrix);
}

void h_matrix::r_multiplication(const string_matrix_type s,bool zero_scaling)
{ node_matrix_type h;
  initialise(h,s); //konvert
  multiplication(matrix,h,zero_scaling);
  delete_matrix(h);
}


const string_matrix_type
                         ROTX={{"1",        "0"        ,"0"        ,"0"},
                               {"0",        "COS(#1)" ,"-SIN(#1)","0"},
                               {"0",        "SIN(#1)" ,"COS(#1)" ,"0"}},
                         ROTY={{"COS(#1)", "0"        ,"SIN(#1)" ,"0"},
                               {"0",        "1"        ,"0"        ,"0"},
                               {"-SIN(#1)","0"        ,"COS(#1)" ,"0"}},
                         ROTZ={{"COS(#1)" ,"-SIN(#1)","0"        ,"0"},
                               {"SIN(#1)" ,"COS(#1)" ,"0"        ,"0"},
                               {"0"       ,"0"       ,"1"        ,"0"}},
                  IDENTITY_M={{"1"        ,"0"       ,"0"        ,"0"},
                               {"0"       ,"1"       ,"0"        ,"0"},
                               {"0"       ,"0"       ,"1"        ,"0"}};
```

Kinematic forward model

```
#ifndef INDIVIDUAL__
#define INDIVIDUAL__
#include "ga_types.h"

#include "my_templates.h"
#include "robot_parameter.h"


class kinematic_type    //represents a forward kinematic robot model
{
public:
  h_matrix           matrix; //matrix containing the symbolic expressions
  parameterlist_type joints; //direct accessable knodes of joint variables
private:
  parameterlist_type parameter;//,linear,nonlinear;//list of free parameters
  void joint_init();

public:
  static int get_integer_from_stdin(char*);
  void init_nominal_parameter();
  kinematic_type(PNode,const string_matrix_type[],int,const tool_type);
  virtual ~kinematic_type();

  double compute_position_error(const dataset_type[],long);
  void   compute_forward_kinematic(dataset_type[],long);
  void print() {parameter.print(stdout);}
};


class kinematic_type_with_derivative: public kinematic_type
{  h_matrix der1,der2,der3,der4,der5;
#if LAST_JOINT==6
   h_matrix der6;
#endif
   h_matrix *derivative[LAST_JOINT];
   int get_joint_with_most_error_m1(const dataset_type[],long,FILE*);
  public:

   kinematic_type_with_derivative(PNode,const string_matrix_type[],int,const
tool_type);
   virtual ~kinematic_type_with_derivative(){}
   int select_joint(const dataset_type[],long,FILE*);
};
#endif


#include "individual.h"
//#include "..\kernel.ok\darstell.h"

//#define WITH_ORIENTATION

#ifdef WITH_ORIENTATION
#define MEASUREMENTS_PER_POSE 6
#else
#define MEASUREMENTS_PER_POSE 3
```

```
#endif

//experimental constructor;
kinematic_type::kinematic_type(PNode                    knode_db,
                               const string_matrix_type robot_link[],
                               int                      link_count,
                               const tool_type          tool)
     :matrix(IDENTITY_M,knode_db),//matrix is initialised with identity
      parameter(100,knode_db),
      joints(6,knode_db)
{ joint_init();
  for (int l=/*1*/0; l<link_count ;l++)
    matrix.r_multiplication(robot_link[l]);
  init_nominal_parameter();
  matrix._add_tool(tool);
}



kinematic_type_with_derivative::kinematic_type_with_derivative(PNode node_db,
                                  const string_matrix_type DH_links[],
                                  int link_number,const tool_type tool_)
       :kinematic_type(node_db,DH_links,link_number,tool_),
        //gradient(MEASUREMENTS_PER_POSE*NUMBER_OF_SAMPLES,LAST_JOINT),
        //error(MEASUREMENTS_PER_POSE*NUMBER_OF_SAMPLES), //vector of samples of
x,y,z errors
        //delta_theta(LAST_JOINT),
        der1(IDENTITY_M,node_db),
        der2(IDENTITY_M,node_db),
        der3(IDENTITY_M,node_db),
        der4(IDENTITY_M,node_db),
        der5(IDENTITY_M,node_db)
#if LAST_JOINT==6
       ,der6(IDENTITY_M,node_db)
#endif
  { derivative[0]=&der1;
    derivative[1]=&der2;
    derivative[2]=&der3;
    derivative[3]=&der4;
    derivative[4]=&der5;
#if LAST_JOINT==6
    derivative[5]=&der6;
#endif
    for (int j=0; j<LAST_JOINT; j++)
    { for (int l=0; l<6 ;l++)
      { if (j==l)
              derivative[j]->r_multiplication(DH_theta_derivative[j],true);
        else  derivative[j]->r_multiplication(DH_links[l]);
      }
      derivative[j]->_add_tool(tool_);
    }
  }



int kinematic_type_with_derivative::select_joint(const dataset_type data[],
                                                 long length,FILE
*logfile)
{

  return get_joint_with_most_error_m1(data,length,logfile);
}



int kinematic_type_with_derivative::get_joint_with_most_error_m1(const dataset_type
data[],
                                                      long  length,FILE
*logfile)
{
    int    j;
    double performance[6]={0.0,0.0,0.0,0.0,0.0,0.0};
    double squared_pose_error[3]={0.0,0.0,0.0};
    double error[3];
    register double z;

    for (int l=0; l<length ;l++)
    {
      joints.assign(data[l].theta);
      matrix.node_set->calculate_list();//new computation of the whole model
      for (int x=0; x<3; x++)
      {  z = (data[l].p[x] - matrix.matrix[x][3]->double_value);
         squared_pose_error[x] += z*z;//fabs(z);//(z*z);
         error[x]=z;
      }
      for (j=0; j<LAST_JOINT; j++)
      { z = 0.0;
        for (int x=0; x<3; x++)
          z+= (error[x] * derivative[j]->matrix[x][3]->double_value);
        performance[j] += fabs(z);
      }
    }
    fprintf(logfile," x: %0.16G  y: %0.16G  z: %0.16G",squared_pose_error[0],
                                                       squared_pose_error[1],
                                                       squared_pose_error[2]);
```

```
    int joint_index=0;
    fprintf(logfile,"  %0.16G",performance[0]);
    for (int x=1; x<LAST_JOINT; x++)
    { fprintf(logfile," %0.16G",performance[x]);
      if (performance[x] > performance[joint_index])
        joint_index = x;
    }
    fprintf(logfile," %i",joint_index);
    return joint_index;
}


#include <float.h>
#define NUMBER_OF_ITERATIONS 10

  kinematic_type::~kinematic_type()
  {}


void kinematic_type::init_nominal_parameter()
{
   for (int l=0; l<18 ;l++)
       parameter.add_to_list(DH_NOMINAL_PARAMETER_[l].name,
                             DH_NOMINAL_PARAMETER_[l].type,
                             DH_NOMINAL_PARAMETER_[l].value);
}

void kinematic_type::joint_init()
{
  char jn[10]="theta1";
  for (int l1=0; l1<6; l1++,jn[5]++) //extracting the joint variable nodes
     joints.add_to_list(jn,NONLINEAR,0);
}



void kinematic_type::compute_forward_kinematic(dataset_type data[],long count)
{
#ifdef WITH_ORIENTATION
  double rot[3];
#endif
  for (int l=0; l<count; l++)
  { joints.assign(data[l].theta);
    matrix.node_set->calculate_list();//new computation of the whole model
#ifdef WITH_ORIENTATION
    euler_angles(matrix.matrix,rot);
#endif
    for (int x=0; x<3; x++)
    {  data[l].p[x]=matrix.matrix[x][3]->double_value;
#ifdef WITH_ORIENTATION
       data[l].p[x+3]=rot[x];
#endif
    }
  }
}

double kinematic_type::compute_position_error(const dataset_type data[],long length)
//distal performance
{ double z=0;
  double z2;
  for (int l=0; l<length ;l++)
  { joints.assign(data[l].theta);
    matrix.node_set->calculate_list();//new computation of the whole model
    for (int x=0; x<3; x++)
    { z2 = data[l].p[x] - matrix.matrix[x][3]->double_value;
      z +=(z2*z2);
    }
  }
  return z;
}

#include <stdlib.h>

int kinematic_type::get_integer_from_stdin(char* s)
{
  printf(s); fflush(stdout);
  char g_buffer[100];
  return atoi(gets(g_buffer));
}
```

Kinematic parameters

```
#ifndef __ROBOT_PARAMETER
#define __ROBOT_PARAMETER
const double PI = 3.1415926535897932385;
typedef double tool_type[3];
extern const tool_type tool;

struct DH_link_parameter
{ double alpha
        ,a
```

```
        ,d;
};

extern const DH_link_parameter PUMA720_parameter[6];

#define LAST_JOINT 6
#define NUMBER_OF_SAMPLES 30

#include "ga_types.h"

extern const variable_type DH_NOMINAL_PARAMETER_[];
extern const string_matrix_type DH_theta_derivative[];
extern const string_matrix_type PUMA_parametric[];//contains the Puma Model to be
calibrated
                                              //for experimental reasons
enum transf_type {ROT_X    ,
                  ROT_Y    ,
                  ROT_Z    ,
                  TRANSL_X ,
                  TRANSL_Y ,
                  TRANSL_Z ,
                  NO_TRANS };

struct elementary_transformation_type
{ char        *name;
  double      Value;
  transf_type type;
};

//for test constructor
//extern const elementary_transformation_type PUMA_parameteric_2[];

#endif


#include "robot_parameter.h"

//main model parameter structure
//all other constant structures further below are defined from them
const DH_link_parameter PUMA720_parameter[6] =
          //  alpha    ,a      ,d
             {{-PI/2   ,0      ,0},
              {0       ,650    ,191},
              {PI/2    ,0      ,0},
              {-PI/2   ,0      ,600},
              {PI/2    ,0      ,0},
              {0       ,0      ,125}};

/* black tool: length  150.25   thickness 20.0025
   aluminium tool      143.536 (hole to hole) 12.93 thickness
                       142.067  hole to top centre (48.617: including tool)

*/
const tool_type tool =
 {150.25,  1.63,    55.69};//measured

#define LINK_PARAMETER(nr,link)\
        {"alpha"#nr,link.alpha,NONLINEAR},\
        {"a"#nr,    link.a,    LINEAR},\
        {"d"#nr,    link.d,    LINEAR}

const variable_type DH_NOMINAL_PARAMETER_[]
  ={LINK_PARAMETER(1, PUMA720_parameter[0]),
    LINK_PARAMETER(2, PUMA720_parameter[1]),
    LINK_PARAMETER(3, PUMA720_parameter[2]),
    LINK_PARAMETER(4, PUMA720_parameter[3]),
    LINK_PARAMETER(5, PUMA720_parameter[4]),
    LINK_PARAMETER(6, PUMA720_parameter[5])};

#define DH_STRING_M(theta,alpha,a,d)\
 {{"cos("theta")", "-cos("alpha")*sin("theta")", "sin("alpha")*sin("theta")" ,
a"*cos("theta")"},\
  {"sin("theta")", "cos("alpha")*cos("theta")" , "-sin("alpha")*cos("theta")",
a"*sin("theta")"},\
  {"0",            "sin("alpha")"             , "cos("alpha")",               d}}

#define DH_THETA_DERIVATIVE_STRING_M(theta,alpha,a,d)\
 {{"-sin("theta")","-cos("alpha")*cos("theta")", "sin("alpha")*cos("theta")", a"*-
sin("theta")"},\
  {"cos("theta")" ,"-cos("alpha")*sin("theta")", "sin("alpha")*sin("theta")",
a"*cos("theta")"},\
  {"0"            ,"0"                         , "0"                        ,"0" }}


const string_matrix_type PUMA_parametric[] //contains the Puma Model to be calibrated
                                           //for experimental reasons
    ={DH_STRING_M("theta1","alpha1","a1" ,"d1"),
      DH_STRING_M("theta2","alpha2","a2" ,"d2"),
      DH_STRING_M("theta3","alpha3","a3" ,"d3"),
      DH_STRING_M("theta4","alpha4","a4" ,"d4"),
      DH_STRING_M("theta5","alpha5","a5" ,"d5"),
      DH_STRING_M("theta6","alpha6","a6" ,"d6")};

const string_matrix_type DH_theta_derivative[]
    ={DH_THETA_DERIVATIVE_STRING_M("theta1","alpha1","a1" ,"d1"),
      DH_THETA_DERIVATIVE_STRING_M("theta2","alpha2","a2" ,"d2"),
```

```
      DH_THETA_DERIVATIVE_STRING_M("theta3","alpha3","a3" ,"d3"),
      DH_THETA_DERIVATIVE_STRING_M("theta4","alpha4","a4" ,"d4"),
      DH_THETA_DERIVATIVE_STRING_M("theta5","alpha5","a5" ,"d5"),
      DH_THETA_DERIVATIVE_STRING_M("theta6","alpha6","a6" ,"d6")};


#define DH_ENT_TRANSFORMATION__(nr,link)\
        {"theta"#nr ,0          ,ROT_Z},\
        {"d"#nr     ,link[nr].d     ,TRANSL_Z},\
        {"a"#nr     ,link[nr].a     ,TRANSL_X},\
        {"alpha"#nr ,link[nr].alpha ,ROT_X}

//for test constructor
const elementary_transformation_type PUMA_parameteric_2[]=
   {DH_ENT_TRANSFORMATION__(1,PUMA720_parameter),
    DH_ENT_TRANSFORMATION__(2,PUMA720_parameter),
    DH_ENT_TRANSFORMATION__(3,PUMA720_parameter),
    DH_ENT_TRANSFORMATION__(4,PUMA720_parameter),
    DH_ENT_TRANSFORMATION__(5,PUMA720_parameter),
    DH_ENT_TRANSFORMATION__(6,PUMA720_parameter)};
```

Expression parsing (lexical and syntactic analysis)

```
/*************************************************************************
   scanner.h

*************************************************************************/
#ifndef _scanner
#define _scanner


#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include "toolunit.h"

#define in_alphabet(ch)      ((toupper(ch)>='A')&&(toupper(ch)<='Z'))
#define in_digits(ch)        (((ch)>='0')&&((ch)<='9'))
#define DefSymbol(sym)       if (strcmp(s,"sym")==0) return sym; else

typedef class generic_scannertyp
{ private:
    int              symbol_length;
    virtual int      next_char()=0;
    static Tsymbole  get_symbol(char);
  protected:
    int              ch;
    const int        EOF_character;
  public :
    symbolstringtyp  symbolstr;
    Tsymbole         symbol;
    bool             read_until_character(char c);
    Tsymbole         read_next_symbol();
    generic_scannertyp(int EOF_char):EOF_character(EOF_char)
    {}
    ~generic_scannertyp(){}
} *PScannertyp;


class Scannertyp: public generic_scannertyp
{ const char       *expression;
  unsigned         expression_index;
  virtual          int next_char();
 public:
  void             take_expression(const char *);
  Scannertyp():generic_scannertyp(0){}
};

#endif


#include "scanner.h"


Tsymbole  generic_scannertyp::get_symbol(char arg)
{
 switch (arg)
   { case   '*':  return MULTIP_TOKEN;
     case   '.':  return DOT;
     case   '(':  return OPEN_PARAN;
     case   ')':  return CLOSE_PARAN;
     case   '+':  return ADD_TOKEN;
     case   '-':  return SUB_TOKEN;
     case   '/':  return DIV_TOKEN;
     case   '#':  return DOUBLECROSS;
     case  '%':   return GP_DIVISION;
   }
   return INVALID;
}


void Scannertyp::take_expression(const char *s)
```

```
{
  expression = s;
  expression_index = 0;
  ch = next_char();
}


int Scannertyp::next_char()
{
  if (expression[expression_index]==0)
      return 0;
  return tolower(expression[expression_index++]);
}



bool generic_scannertyp::read_until_character(char c)
{ symbol_length=0;
  while ( ch != EOF_character)
    if (ch==c)
      { ch=next_char();
        symbolstr[symbol_length]=0;
        return true;
      }
      else
        { if (symbol_length < max_token_length)
            symbolstr[symbol_length++]=ch;
          ch = next_char();
        }
  symbol=ENDTOKEN;
  return false;
}

Tsymbole generic_scannertyp::read_next_symbol()
{ symbol_length = 0;
  while ((ch==' '))
      ch = next_char();
  if (ch == EOF_character) { symbol=ENDTOKEN;return ENDTOKEN;}
  if (in_alphabet(ch))
  { symbol=IDENTIFIER;
    do   { if (symbol_length < max_token_length)
             symbolstr[symbol_length++] = ch ;
           ch = next_char();
         } while (in_alphabet(ch)||(in_digits(ch)));
  }
    else if (in_digits(ch))
         { if (symbol_length==0) symbol=NUM_TOKEN;
           do
           { if (symbol_length < max_token_length)
             symbolstr[symbol_length++] = ch;
             ch = next_char();
           } while (in_digits(ch));
         }
  if (symbol_length==0)  // special character
  {   symbolstr[symbol_length++]=ch;
      ch=next_char();
      symbol= get_symbol(symbolstr[0]);
      return symbol;
  }
  symbolstr[symbol_length]=0;       //termination
  return symbol;
}

#ifndef _expr_parser
#define _expr_parser
/*************************************************************************
   parser.h
   Jens- Uwe Dolinsky

   Simple expression parser used for establishing kinematic equations
*************************************************************************

EBNF
  CHARACTER      := 'a'..'z' | 'A'..'Z'.
  DIGIT          := '0'..'9'.
  IDENT          := CHARACTER {CHARACTER|DIGIT}.
  NUMBER         := DIGIT {DIGIT}.
  NUM            := '.' NUMBER
                    | NUMBER ['.' NUMBER] [factor].
  operand        := NUM
                    | IDENT ['(' sum ')'].
  factor         := ('(' sum ')')
                    | operand.
  signed_factor  := {'-'|'+'} (factor).
  term           := signed_factor { ('*'|'/') signed_factor}.
  sum            := term {'+'|'-'} term }.

  start_symbol   := sum.
*/

#include "knot_tab.h"
#include "scanner.h"


typedef class Parsertyp : private Scannertyp
         {
          private:
```

```
            PNode Kdb; //Node set
            PNode get_node(const char*,Tsymbole,PNode,PNode);
          virtual PNode read_operand()        ;//throw(String);
                  PNode read_signed_factor()  ;//throw(String);
                  PNode read_factor()         ;//throw(String);
                  PNode read_term()  ;//throw(String);
                  PNode read_sum() ;//throw(String);
                  void    throw_error(char*)         ;//throw(String);
          public:
            PNode read_expression(char*,PNode);//throw(String);
        } *PParsertyp;

#endif

#include "parser.h"
#include <stdlib.h>


PNode Parsertyp::get_node(const char* name,Tsymbole sym,PNode left,PNode right)
{ PNode h = Kdb->get_node(name,sym,left,right);
  h->create_reference();
  referenztyp::remove_one_reference(h->lnext);
  referenztyp::remove_one_reference(h->rnext);
  return h;
}


PNode Parsertyp::read_operand()
{ PNode          h=NULL;
    symbolstringtyp  sym;
    sym[0]=0;
    switch(symbol)
    { case NUM_TOKEN:      h= get_node(symbolstr,NUM_TOKEN,NULL,NULL);
                       read_next_symbol();
                       return h;
      case IDENTIFIER:
                        strcat(sym,symbolstr);
                        read_next_symbol();
                        if (symbol==OPEN_PARAN)
                        { h = read_sum();
                          if (symbol!=CLOSE_PARAN)
                                throw_error(") expected");
                             else h = get_node(sym,FUNC_TOKEN,NULL,h);
                          read_next_symbol();
                        } else
                           if (Node::compare_identifiers(sym,"pi")==0)
                              h = get_node(sym,STRING_CONST,NULL,NULL);
                            else h = get_node(sym,VARIABLE,NULL,NULL);
                        return h;
    }
    throw_error("need identifier or number");
    return h;
}


void Parsertyp::throw_error(char* s)
{   throw string_class(s);
}


PNode Parsertyp::read_factor()
{PNode h=NULL,h1;
 switch(symbol)
    { case OPEN_PARAN: h = read_sum();
                       if (symbol!=CLOSE_PARAN) throw_error(") expected");
                       read_next_symbol();
                       break;
      default: h = read_operand();
    }
 return h;
}


PNode Parsertyp::read_signed_factor()
{ PNode h=NULL;
  Tsymbole operation = ADD_TOKEN;
  while (dash_operator(symbol))
  { operation = (operation==symbol)?ADD_TOKEN:SUB_TOKEN;
    read_next_symbol();
  }
  h=read_factor();
  if (operation==SUB_TOKEN)   //negative sign
     {
       PNode h1 = get_node("0",NUM_TOKEN,NULL,NULL);
       h = get_node(NULL,SUB_TOKEN,h1,h);
     }
 return h;
}


PNode Parsertyp::read_term()
{  PNode  h=NULL,h1;
   Tsymbole hsymbol;
     h = read_signed_factor();
     while (dot_operator(symbol))
     {  hsymbol=symbol;
```

```
        read_next_symbol();
        h1 = read_signed_factor();
        h =  get_node(NULL,hsymbol,h,h1);
     }
   return h;
}

PNode Parsertyp::read_sum()
{ PNode h=NULL,h1;
  Tsymbole operation ;
  read_next_symbol();
    h = read_term();
    while (dash_operator(symbol))
    { operation = ADD_TOKEN;
      do { operation = (operation == symbol)?ADD_TOKEN:SUB_TOKEN;
           read_next_symbol();
         } while (dash_operator(symbol));
      h1 = read_term();
      h = get_node(NULL,operation,h,h1);
    }
  return h;
}


PNode Parsertyp::read_expression(char *s,PNode menge)
{
  PNode h=NULL;
  take_expression(s);
  Kdb=menge;
  h = read_sum();
  if (symbol!=ENDTOKEN)
       throw_error("end of expr. expected, but found more");
  return h;
}
```

Templates for matrices and vectors

```
#ifndef _MY_TEMPLATES
#define _MY_TEMPLATES

template <class type__> class m_vector
    //    :public raw_vector<type__>
{ public:
    const  int dimension;
    type__ *data;
    m_vector(int d):dimension(d)
    {
        data=new type__[d];
    }
    ~m_vector()
    {
        delete[]data;
    }
    void assign(m_vector<type__> &v)
    { for (int l=0; l<dimension; l++)
        data[l]=v.data[l];
    }

};


template <class type__> class matrix_template
{public:
  type__ **data;
  const int rows,columns,size_;

  matrix_template(int r,int c)
    :rows(r),columns(c),size_(r*c)
  { data= (type__**)new type__[r];
    for (int l=0; l<rows; l++)
       data[l]= new type__[c];
  }
  ~matrix_template()
  { for (int l=0; l<rows; l++)
        delete []data[l];
    delete []data;
  }
  void assign(int r,int c,type__ value)
  { //if ((r>=rows) || (c>=columns))
      // printf("matrixdimension exceeded");
    data[r][c]=value;
  }
  type__ value(int r, int c)
  { //if ((r>=rows) || (c>=columns))
      // printf("matrixdimension exceeded");
    return data[r][c];
  }
  void swap_rows(const int r1,const int r2)
  { type__ z;
    for (register int l=0; l<columns; l++)
    { z= data[r1][l];
      data[r1][l]=data[r2][l];
```

```
      data[r2][l]=z;
    }
  }

};

#endif
```

Local frames

```
#ifndef __LOCAL_FRAME
#define __LOCAL_FRAME


typedef double vector3D[3];
typedef vector3D HG_matrix_type[4];
#include <stdio.h>
int get_measured_data(const char *robotrak_local_frame_file,
                      const char *robot_local_frame_file,
                      const char *robotrak_datafile,
                      const char *robot_datafile,
                      const char *error_diff_file,
                      vector3D  robotrak_measurements[],
                      vector3D  robot_measurements[],
                      const char* taskspace_file);
#include "ga_types.h"

void create_program_file(const char * name,const dataset_type config[],const int
number);

void create_lc_program_file(const char          *name,
                            const dataset_type  config[],
                            const int           number,
                            class kinematic_type*ptr_kinematic_model,
                            const char          *location_filename);


typedef double jointangle_set[6];
int get_joint_angles(const char *filename,jointangle_set j[]);


void puma_forward(const dataset_type[],const int,const char*,const vector3D);
void puma_inverse(const double          config[],
                  const HG_matrix_type data,
                  const int            lockwrist,
                  double               theta[],
                  const vector3D       tool);

const HG_matrix_type HG_IDENTITY_M = {{1,0,0},{0,1,0},{0,0,1},{0,0,0}};

#endif
```

Transformation routines and parsing of VALII files

```
#ifndef __FILE_SCANNER
#define __FILE_SCANNER
#include "scanner.h"

class FileScannertyp: public generic_scannertyp //string scanner
{ FILE *stream;
  virtual int next_char()
  { int ch= fgetc(stream);
    while ((ch==10)||(ch==13))
      ch= fgetc(stream);
    return ch;
  }
 public:
  FileScannertyp(const char *f):generic_scannertyp(EOF/*EOF character*/)
  { stream=fopen(f,"rb");
    if (stream==NULL)
      throw int(1);
  }
  ~FileScannertyp()
  { if (stream!=NULL)
      fclose(stream);
  }
};
#endif



#ifndef __data_set__
#define __data_set__

#include "individual.h"

extern dataset_type teachpoints__[];
void convert_to_rad(dataset_type &);
void prepare_datasets(kinematic_type&,dataset_type[],int);
void generate_datasets(dataset_type[],const long,const long,const tool_type);
```

```
#endif



#include "dataset.h"


void convert_to_rad(dataset_type &joints)
{ for (int l=0; l<6; l++)
    joints.theta[l]=(joints.theta[l]*PI)/180;
}


#include "local_frame.h"


#define SCALING_FACTOR 16

static void vector_assign(const vector3D &a,vector3D &r)
{   r[0]=a[0];
    r[1]=a[1];
    r[2]=a[2];
}

static void vector_product(const vector3D &a,const vector3D &b,vector3D &ab)
{   //VECTOR([a1*b2-a2*b1, a2*b0-a0*b2, a0*b1-a1*b0])
    ab[0]=a[1]*b[2]-a[2]*b[1];
    ab[1]=a[2]*b[0]-a[0]*b[2];
    ab[2]=a[0]*b[1]-a[1]*b[0];
}

static void vector_sub(const vector3D &a,const vector3D &b,vector3D &ab)
{   ab[0]=a[0]-b[0];
    ab[1]=a[1]-b[1];
    ab[2]=a[2]-b[2];
}

static double scalar_product(const vector3D &a,const vector3D &b)
{ return a[0]*b[0]+
         a[1]*b[1]+
         a[2]*b[2];
}

#include <math.h>
static void norm_vector(const vector3D &a,vector3D &na)
{   double n=sqrt(scalar_product(a,a));
    na[0]=a[0]/n;//x axis is normalised ab
    na[1]=a[1]/n;
    na[2]=a[2]/n;
}

static void frame_multiplication(const HG_matrix_type &left,
                                 const HG_matrix_type &right,
                                       HG_matrix_type &result)
{
  for (int r=0; r<3; r++)
  { for (int c=0; c<4; c++)
    { //element with first product
      result[c][r] = 0;//left[0][r]*right[c][0];
      for (int l=0;l<3;l++)//adding all products
        result[c][r]+= left[l][r]*right[c][l];
    }
    result[3][r]+=left[3][r];
  }
}

static void transform_lf(const HG_matrix_type lf,const vector3D pose, vector3D
&result)
{  for (int r=0; r<3; r++)
   { result[r]=0.0;
     for (int c=0; c<3; c++)
       result[r]+=(lf[c][r]*pose[c]);
     result[r]+=lf[3][r];//homogenous coordinates
   }
}

static void inverse(const HG_matrix_type& m,HG_matrix_type& res)
{ for (int c=0; c<3; c++)
    for (int r=0; r<3; r++)
      res[c][r] = m[r][c];

  vector3D ori;
  for (int r=0; r<3; r++)
  { ori[r]=0;
    for (int c=0; c<3; c++)
      ori[r]+=(res[c][r]*m[3][c]);
    ori[r]=-ori[r];
  }
  vector_assign(ori,res[3]);
}


void create_local_frame(const vector3D &a,
                        const vector3D &b,
```

```
                        const vector3D &c,
                        const vector3D &org,
                        HG_matrix_type &local_frame)
{   // ->
    // ab = a- b
    vector3D ab;
    vector_sub(b,a,ab);
    vector3D normed_ab;
    norm_vector(ab,normed_ab); //x axis is normalised ab

    vector3D ac;
    vector_sub(c,a,ac);
    double lambda =scalar_product(ac,normed_ab);
    vector3D aE;
      aE[0]=lambda*normed_ab[0];
      aE[1]=lambda*normed_ab[1];
      aE[2]=lambda*normed_ab[2];
    vector3D Ec;
    vector_sub(ac,aE,Ec);
    vector3D normed_Ec;
    norm_vector(Ec,normed_Ec); // y axis

    vector_assign(normed_ab,local_frame[0]);
    vector_assign(normed_Ec,local_frame[1]);
    vector_product(normed_ab,normed_Ec,local_frame[2]); //z axis
    vector_assign(org,local_frame[3]);
}


static void create_inv_local_frame(
                        const vector3D &a,
                        const vector3D &b,
                        const vector3D &c,
                        const vector3D &org,
                        HG_matrix_type &inv_transformation)
{   HG_matrix_type local_frame;
    create_local_frame(a,b,c,org,local_frame);
    inverse(local_frame,inv_transformation);
}



#include <stdio.h>
#include <stdlib.h>
#include "file_scanner.h"

static double read_double(FILE *f)
{ char buffer[200];
  int ch;
  for(;;)
  { switch(ch=fgetc(f))
    { case EOF: throw int(0);
      case ' ':
      case '\n': continue;
      default: break;
    }
    break;
  }
  int l=0;
  for (;;)
  { buffer[l++]=ch;
    switch(ch=fgetc(f))
    { case EOF:
      case ' ':
      case '\n': break;
      default:  continue;
    }
    break;
  }
  buffer[l]=0;
  return atof(buffer);
}

static void read_robotrak_local_frame_data(const char* file,HG_matrix_type &m)
{ FILE *f=fopen(file,"r");
  if (f==NULL)
  {   printf("Cannot open file %s",file);
      throw int(0);
  }
  for (int c=0; c<4; c++)
    for (int r=0; r<3; r++)
      (m[c])[r]=0.0;
  for(int counter=0;;counter++)
  {  for (int c=0; c<4; c++)
       for (int r=0; r<3; r++)
         try { double p=read_double(f);
               m[c][r] += p;
             }
         catch(...)
         { if ((c!=0)||(r!=0)||(counter==0))
           {  printf("problems reading local framepoints");
              fclose(f);
              throw int(0);
           }
            else
```

```
              for (int c=0; c<4; c++)
                for (int r=0; r<3; r++)
                  m[c][r]/=counter;
            fclose(f);
            return;
          }
  }
}

static void get_robotrak_inv_local_frame(const char *name, HG_matrix_type &lf)
{ HG_matrix_type m;
  read_robotrak_local_frame_data(name,m);
  create_inv_local_frame(m[0],m[1],m[2],m[3],lf);
}

static int read_integer(FileScannertyp &sc)
{ bool minus;
  if (sc.read_next_symbol()==SUB_TOKEN)
  {  minus=true;
     sc.read_next_symbol();
  } else minus=false;
  if (sc.symbol!=NUM_TOKEN)
  { printf("\nFile format error: expecting number");
    throw int(0);
  }
  return (minus==true)?-atoi(sc.symbolstr):atoi(sc.symbolstr);
}

static double read_double(FileScannertyp &sc)
{ auto char buffer[80]="";
  if (sc.read_next_symbol()==SUB_TOKEN)
  {  sc.read_next_symbol();
     strcpy(buffer,"-");
  }
  if (sc.symbol!=NUM_TOKEN)
  { printf("\nFile format error: expecting number");
    throw int(0);
  }
  strcat(buffer,sc.symbolstr);
  if (sc.read_next_symbol()!=DOT)
  { printf("\nFile format error: expecting .");
    throw int(0);
  }
  strcat(buffer,".");
  if (sc.read_next_symbol()!=NUM_TOKEN)
  { printf("\nFile format error: expecting number");
    throw int(0);
  }
  strcat(buffer,sc.symbolstr);
  return atof(buffer);
}


typedef int pose_type[3];

static bool read_next_pose(FileScannertyp &sc,pose_type &p, int *tr_m)
{ int l;
  while (sc.symbol==DOUBLECROSS)
  { sc.read_until_character(' ');
    for (l=0;l<6;l++)
      read_double(sc);
    sc.read_next_symbol();
  }
  if (sc.symbol!=IDENTIFIER)
    return false;
  sc.read_until_character(' ');
  for (l=0; l<9; l++)
    if (tr_m==NULL) read_integer(sc);//overread first 9 numbers
      else tr_m[l] = read_integer(sc);
  for (l=0; l<3; l++)
    p[l]=read_integer(sc);
  return true;
}


static void read_until_location_points(FileScannertyp &scanner)
{ while (scanner.read_next_symbol()!=ENDTOKEN)
    if (scanner.symbol==DOT)
      if (scanner.read_next_symbol()==IDENTIFIER)
        if (strcmp(scanner.symbolstr,"LOCATIONS")==0)
          return;
  printf("\n No locations found");
  throw int(0);
}


static void read_robot_local_frame(const char *filename,HG_matrix_type &lf)
{ FileScannertyp sc(filename);
  read_until_location_points(sc);
  int  f[3][4];
  for (int l=0;l<3;l++)
    for (int k=0;k<4;k++)
      f[l][k]=0;
  for (int counter=0;;counter++)
  { for(int k=0;k<4;k++)
    { if (sc.read_next_symbol()!=IDENTIFIER)
      { if (counter==0)
```

```
        { printf("\n<robot local frame> file format error: expect location point
name");
          throw int(0);
        }
        for (int l=0;l<3;l++)
          for (int k1=0;k1<4;k1++)
            lf[k1][l] = (double)f[l][k1]/(SCALING_FACTOR*counter);
        return;
      }
      pose_type p;
      if (read_next_pose(sc,p,NULL)==false)
      { printf("\nCannot read location point");
        throw int(0);
      }
      for (int l=0; l<3; l++)
         f[l][k] += p[l];
    }
  }
}

//static void get_robot_local_frame(const char *filename,HG_matrix_type &lf)

static void get_robot_local_frame(const char *filename,HG_matrix_type &lf)
{ HG_matrix_type local;
  read_robot_local_frame(filename,local);
  create_local_frame(local[0],local[1],local[2],local[3],lf);
}
/*
static void get_robot_inv_local_frame(const char *filename,HG_matrix_type &lf)
{ HG_matrix_type local;
  read_robot_local_frame(filename,local);
  create_inv_local_frame(local[0],local[1],local[2],local[3],lf);
}*/


int get_joint_angles(const char *filename,jointangle_set j[])
{ int counter=0;
  FileScannertyp scanner(filename);
  read_until_location_points(scanner);
        for(;;)
            switch (scanner.read_next_symbol())
            { case ENDTOKEN:  return counter;
              case DOUBLECROSS:{ scanner.read_until_character(' ');
                           for (int l=0; l<6; l++)
                            j[counter][l]=read_double(scanner);
                           counter++;
                           continue;
                         }
              default:continue;
            }
}

static int round(double v)
{
  v+=0.5;
  return (int)v;
}

int convert_locations(const char *filename,const char *outp,
                      const vector3D robot[],const vector3D rtrack[])
{
  FileScannertyp sc(filename);
  read_until_location_points(sc);
  pose_type p;
  int tr[9];
  FILE *f=fopen(outp,"w+");
  if (f==NULL) return -1;
  fprintf(f,".PROGRAM correct\n"
            "FOR l = 1 TO %u\n"
            "MOVE mcp[l]\n"
            "DELAY 4\n"
            "HERE #cp[l]\n"
            "END\n"
            ".END\n"
            ".LOCATIONS\n",30);
  for(int counter=0;;counter++)
  { sc.read_next_symbol();
    if (read_next_pose(sc,p,tr)==false)
    {   fprintf(f,".END\n");
        fclose(f);
        return counter;
    }
    fprintf(f,"mcp[%u]  ",counter+1);
    for (int pc=0; pc<9; pc++)
      fprintf(f,"%i ",tr[pc]);
    for (int l=0; l<3; l++)//data[l].p[j] -= ( robotrak[l][j]-robot_poses[l][j] );
      fprintf(f,"%i ",p[l]-round((rtrack[counter][l]-
robot[counter][l])*SCALING_FACTOR));
    fprintf(f,"\n");
  }
}


static int read_and_transform_robotlocations(
                                      const char           *filename,
                                      const HG_matrix_type &lf,
                                      vector3D             poses[])
```

```
{ FileScannertyp sc(filename);
  read_until_location_points(sc);
  pose_type p;
  vector3D  double_pose;
  for(int counter=0;;counter++)
  { sc.read_next_symbol();
    if (read_next_pose(sc,p,NULL)==false)
      return counter;
    for (int l=0; l<3; l++)
    {  double_pose[l] = p[l];
       double_pose[l]/=SCALING_FACTOR;
    }
    /*if (lf==NULL)
     for (int p=0; p<3; p++)
        poses[counter][p]=double_pose[p];
      else*/ transform_lf(lf,double_pose,poses[counter]);
  }
}

static int read_and_transform_robotrack_locations(
                                          const char           *filename,
                                          const HG_matrix_type &lf,
                                          vector3D             poses[])
{
  FILE *f=fopen(filename,"r");
  if (f==NULL)
  {   printf("Cannot open file %s",filename);
      throw int(0);
  }
  vector3D  double_pose;
  for(int counter=0;;counter++)
  { for (int c=0; c<3; c++)
    { try { double_pose[c]=read_double(f); }
      catch(...)
      { if (c!=0)
        { printf("format error in file %s",filename);
          fclose(f);
          throw int(0);
        }
        fclose(f);
        return counter;
      }
    }
    transform_lf(lf,double_pose,poses[counter]);
  }
}


double write_error_file(
                        const vector3D    robotrak_measurements[],
                        const vector3D    robot_measurements[],
                        int         number,
                        const char* filename)
{ FILE *f=fopen(filename,"w+");
  if (f==NULL)
  {  printf("\nCould not write error statistic file");
     return -1;
  }
  double total=0.0;
  double m_error,difference;
  for (int l=0; l<number; l++)
  { m_error=0.0;
    for (int k=0; k<3; k++)
    { difference = robotrak_measurements[l][k] - robot_measurements[l][k];
      m_error+= (difference*difference);//fabs(difference);
      fprintf(f,"%lf ",difference);
    }
    fprintf(f," %lf\n",sqrt(m_error));
    total+=m_error;
//    fprintf(f,"%lf\n",m_error);
  }
  fprintf(f,"total squared :%lf\n",total);
  fclose(f);
  return total;
}


int get_measured_data(const char *robotrak_local_frame_file,
                      const char *robot_local_frame_file,
                      const char *robotrak_datafile,
                      const char *robot_datafile,
                      const char *error_diff_file,
                      vector3D  robotrak_measurements[],
                      vector3D  robot_measurements[],
                      const char* taskspace_file)//,

{
   try
   {
      HG_matrix_type rm;
        get_robot_local_frame(robot_local_frame_file,rm);
      HG_matrix_type rt_m;
        get_robotrak_inv_local_frame(robotrak_local_frame_file,rt_m);
      HG_matrix_type transf;
      frame_multiplication(rm,rt_m,transf);
```

```
      int l
=read_and_transform_robotrack_locations(robotrak_datafile,transf,robotrak_measurement
s);
      for (int r=0; r<3; r++)
      {  transf[3][r]=0;
         for (int c=0; c<3; c++)
           transf[c][r]=(c==r)?1:0;
      }
      if
(l!=read_and_transform_robotlocations(robot_datafile,transf,robot_measurements))
      { printf("nonequal number of measurements");
        return -1;
      }
      if (error_diff_file!=NULL)

write_error_file(robotrak_measurements,robot_measurements,l,error_diff_file);


//write_error_file(transf_robotrak_measurements,exact_robot_poses,l,error_diff_file);
      //
write_error_file(robotrak_measurements,robot_measurements,l,"c:\\data\\cmp1.txt");
      //
write_error_file(transf_robotrak_measurements,exact_robot_poses,l,"c:\\data\\cmp2.txt
");

convert_locations(robot_datafile,taskspace_file,robot_measurements,robotrak_measureme
nts);

      return l;
   }
   catch(...)
   { return -1;
   }
}


/*
int get_measured_data(const char *robotrak_local_frame_file,
                      const char *robot_local_frame_file,
                      const char *robotrak_datafile,
                      vector3D  robotrak_measurements[])
{
   try
   {  HG_matrix_type inv_rt;
      get_robotrak_inv_local_frame(robotrak_local_frame_file,inv_rt);
      HG_matrix_type robotm;
      get_robot_local_frame(robot_local_frame_file,robotm);
      HG_matrix_type transf;
      frame_multiplication(robotm,inv_rt,transf);
      return read_and_transform_robotrack_locations(robotrak_datafile,
                                                    transf,robotrak_measurements);
   }
   catch(...)
   { return -1;
   }
}*/

#include "ga_types.h"
#include "robot_parameter.h"

void create_program_file(const char * name,const dataset_type config[],const int
number)
{ FILE *f=fopen(name,"w+");
  if (f==NULL)
  { printf("cannot open file %s for writing",name);
    return;
  }
  fprintf(f,".PROGRAM correct\n"
            "FOR l = 1 TO %u\n"
            "MOVE #cp[l]\n"
            "DELAY 2\n"
            "TYPE \"stop \", l\n"
            "DELAY 3\n"
            "END\n"
            ".END\n"
            ".LOCATIONS\n",number);
  for (int l=0; l<number; l++)
  { fprintf(f,"#cp[%u]",l+1);
    for (int j=0; j<6; j++)
      fprintf(f," %5.6lf",(config[l].theta[j]*180/PI));
    fprintf(f,"\n");
  }
  fprintf(f,".END\n");
  fclose(f);
}


#include "individual.h"

void create_lc_program_file(const char          *name,
                            const dataset_type  config[],
                            const int           number,
                            kinematic_type     *ptr_kinematic_model,
                            const char          *location_filename)
{ FILE *f=fopen(name,"w+");
  if (f==NULL)
```

```
  { printf("cannot open file %s for writing",name);
    return;
  }
  fprintf(f,".PROGRAM correct\n"
            "FOR l = 1 TO %u\n"
            "MOVE mcp[l]\n"
            "DELAY 2\n"
            "TYPE \"stop \", l\n"
            "DELAY 3\n"
            "END\n"
            ".END\n"
            ".LOCATIONS\n",number);
  FileScannertyp sc(location_filename);
  read_until_location_points(sc);
  pose_type p;
  int tr_m[9];
//  vector3D  double_pose;
  sc.read_next_symbol();
  if (read_next_pose(sc,p,tr_m)==false)
  { printf("\n no locations");
    throw int(0);
  };
  dataset_type ds;
  for (int l=0; l<number; l++)
  { for (int j=0; j<6; j++)
      ds.theta[j] = config[l].theta[j];
    ptr_kinematic_model->compute_forward_kinematic(&ds,1);
    fprintf(f,"mcp[%u]",l+1);
    for (int t=0; t<9; t++)
      fprintf(f," %i",tr_m[t]);
    for (int k=0; k<3; k++)
      fprintf(f," %i",round(ds.p[k]*16));
    fprintf(f,"\n");
  }
  fprintf(f,".END\n");
  fclose(f);
}


void generate_matrix(const double theta[],
                     const double aa[],
                     const double a[],
                     const double d[],
                     const int i,
                     HG_matrix_type &A)
{ A[0][0]= cos(theta[i]);
  A[1][0]=-sin(theta[i])*cos(aa[i]);
  A[2][0]= sin(theta[i])*sin(aa[i]);
  A[3][0]= a[i]*cos(theta[i]);

  A[0][1]= sin(theta[i]);
  A[1][1]= cos(theta[i])*cos(aa[i]);
  A[2][1]=-cos(theta[i])*sin(aa[i]);
  A[3][1]=a[i]*sin(theta[i]);

  A[0][2]=0;
  A[1][2]=sin(aa[i]);
  A[2][2]=cos(aa[i]);
  A[3][2]=d[i];
}
```

Main program routines

```
#include "gp_system.h"

#define DATA_DIR_ "f:\\workdir\\"


class file_manager //just for controlling the resource (file) aquisition
{ public:
  FILE *file;
  file_manager(const char*name,const char* attr)
  { file=fopen(name,attr);
    if (file==NULL)
     printf("cannot open file: %s",name);
  }
  ~file_manager()
  { if (file!=NULL) fclose(file);
  }
};


#include "local_frame.h"

int get_and_convert_joint_angles(const char         *teachpoint_file,
                                 dataset_type        data[],
                                 jointangle_set      joint_values[])
{ int n2=get_joint_angles(teachpoint_file,joint_values);
  if (n2<1)
  {  printf("\nCannot find %s to read joint angles",teachpoint_file);
```

```
     return -1;
  }
  for (int l=0; l<n2; l++)
  { for (int x=0; x<6; x++)
       data[l].theta[x] = joint_values[l][x];
    convert_to_rad(data[l]);
  }
  return n2;
}


int initialise_teachpoints(dataset_type data[],kinematic_type *ptr_kinematic_model)
{ jointangle_set joint_values[160];
  vector3D      robotrak[160],robot_poses[160];//,tr_robotrak[100],ex_robposes[100];
 // test_transformations();
  try
  { //write corrected differences if available
    get_measured_data(DATA_DIR "Lfs.ext",//robotrak_local_frame_file,
                      DATA_DIR "lfs.v2",//const char *robot_local_frame_file,
                      DATA_DIR "correctt1.ext",//const char *robotrak_datafile,
                      DATA_DIR "t1.v2",//const char *robot_datafile,
                      DATA_DIR "corrected_differences_t1.txt",
                      robotrak,robot_poses/*,tr_robotrak,ex_robposes*/,
                      DATA_DIR "tko.v2");
    get_measured_data(DATA_DIR "Lfs.ext",//robotrak_local_frame_file,
                      DATA_DIR "lfs.v2",//const char *robot_local_frame_file,
                      DATA_DIR "correctt2.ext",//const char *robotrak_datafile,
                      DATA_DIR "t2.v2",//const char *robot_datafile,
                      DATA_DIR "corrected_differences_t2.txt",
                      robotrak,robot_poses/*,tr_robotrak,ex_robposes*/,
                      DATA_DIR "tko.v2");
   get_measured_data(DATA_DIR "Lfs.ext",//robotrak_local_frame_file,
                      DATA_DIR "lfs.v2",//const char *robot_local_frame_file,
                      DATA_DIR "t2.ext",//const char *robotrak_datafile,
                      DATA_DIR "t2.v2",//const char *robot_datafile,
                      DATA_DIR "t2_differences.txt",
                      robotrak,robot_poses/*,tr_robotrak,ex_robposes*/,
                      DATA_DIR "t2_ts_sp_correct.v2");
   int n=get_measured_data(DATA_DIR "Lfs.ext",//robotrak_local_frame_file,
                           DATA_DIR "lfs.v2",//const char *robot_local_frame_file,
                           DATA_DIR "t1.ext",//const char *robotrak_datafile,
                           DATA_DIR "t1.v2",//const char *robot_datafile,
                           DATA_DIR "differences.txt",
                           robotrak,robot_poses/*,tr_robotrak,ex_robposes*/,
                           DATA_DIR "t1_ts_sp_correct.v2");

   int n2=get_joint_angles(DATA_DIR "t1.v2",joint_values);
   if ((n==-1)||(n2==-1)) return -1;
   if (n!=n2)
   { printf("mismatch: number of poses and jointangles");
     return -1;
   }
   int j;
   for (int l=0;l<n;l++)
   { for (j=0; j<6; j++)
       data[l].theta[j] = joint_values[l][j];
     convert_to_rad(data[l]);
     ptr_kinematic_model->compute_forward_kinematic(&data[l],1);
     for (j=0; j<3; j++)
     {  //data[l].p[j] = tr_robotrak[l][j];
        data[l].p[j] -= ( robotrak[l][j]-robot_poses[l][j] );
     }
   }
   return n;
 }
  catch(...)
  {}
  return -1;
}


void alter_joint_angles(const char* source_file,
                        const char* dest_file,
                        abstract_gp_system *msystem)
{ jointangle_set joint_values[160];
  dataset_type   data[160];
  try
  {  int n2=get_joint_angles(source_file,joint_values);
     if (n2<1)
     {  printf("\nCannot alter file: %s",source_file);
        return;
     }
     for (int l=0; l<n2; l++)
     {  for (int j=0; j<6; j++)
          data[l].theta[j] = joint_values[l][j];
        convert_to_rad(data[l]);
     }
    msystem->correct(data,n2);
    create_program_file(dest_file,data,n2);
  }
  catch(...)
  { printf("\n Unknown error while altering joint values of: %s",source_file);
  }
}
```

```
#include <time.h>
#include <stdlib.h>

void gp_direct_joint_error_learning(PNode node_db) //direct joint error learning
{
  node_db->create_reference();
  try
  { kinematic_type kinematic_model(node_db,PUMA_parametric,6,tool);
    dataset_type data[100];
    int data_samples=initialise_teachpoints(data,&kinematic_model);
    if (data_samples==-1) return;
    srand(time(0)); //initialising the random number generator
    file_manager logfile(DATA_DIR_"gp_logfile.txt","w+");
    jointangle_set joint_values[160];
    int df =
get_and_convert_joint_angles(DATA_DIR_"t1correct.v2",&data[data_samples],joint_values
);
    if (df!=data_samples) return;
    calibration_system
robotgp_system(data,data_samples,kinematic_model);//instantiate system
    robotgp_system.ga(data,data_samples,logfile.file);//start evolution
    alter_joint_angles(DATA_DIR_"t1.v2",DATA_DIR_"updated_t1.v2",&robotgp_system);
    alter_joint_angles(DATA_DIR_"t2.v2",DATA_DIR_"updated_t2.v2",&robotgp_system);
    robotgp_system.write_statistic(logfile.file);
  }
  catch(int l)
  { printf("\n%u ",l);
    switch(l)
    {case  9: printf("MATH_NOT_EVALUABLE ");break;
     case 10: printf("ACCESS_VIOLATION");break;
     default:printf("Unknown Error");
    }
  }
  catch(...)
  { printf("Unexpected Exception\n");
  }
  Node::remove_one_reference(node_db);


}

void distal_supervised_learning(PNode node_db)
{
  node_db->create_reference();
  try
  { kinematic_type kinematic_model(node_db,PUMA_parametric,6,tool);
    dataset_type data[100];
    int data_samples=initialise_teachpoints(data,&kinematic_model);
    if (data_samples==-1) return;
    srand(time(0)); //initialising the random number generator
    file_manager logfile(DATA_DIR_"gp_logfile.txt","w+");
    calibration_system robotgp_system(data,data_samples,kinematic_model);
    robotgp_system.ga(data,data_samples,logfile.file);
    alter_joint_angles(DATA_DIR_"t1.v2",DATA_DIR_"updated_t1.v2",&robotgp_system);
    alter_joint_angles(DATA_DIR_"t2.v2",DATA_DIR_"updated_t2.v2",&robotgp_system);
    robotgp_system.write_statistic(logfile.file);
  }
  catch(int l)
  { printf("\n%u ",l);
    switch(l)
    {case  9: printf("MATH_NOT_EVALUABLE ");break;
     case 10: printf("ACCESS_VIOLATION");break;
     default:printf("Unknown Error");
    }
  }
  catch(...)
  { printf("Unexpected Exception\n");
  }
  Node::remove_one_reference(node_db);
}


void main_gp2(PNode node_db)
{
#ifdef PARALLEL_MODELLING
  gp_direct_joint_error_learning(node_db);
#else
  distal_supervised_learning(node_db);
#endif
}



void main()
{ PNode k = Node::createNode("pi",STRING_CONST,NULL,NULL);
  main_gp2(k);
  Node::remove_one_reference(k);
}
```

References

[1] ABB. (2000). *RobotStudio – Programming & Simulation*. RobotStudio® product description, Hannover 2000.

[2] Adby, P.R. and M.A.H. Dempster. (1974). *Introduction To Optimization Methods*. Chapman & Hall, London.

[3] Aho A. V., R. Sethi & J. D. Ullman. (1986). *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

[4] Albright S. and K. Schröer. (1992*). Practical Error Compensation for Use in Off-Line Programming of Robots*. In Robotic Systems: Advanced Techniques and Applications (ed. Tzafestas), Kluwer Academic Publishers, pp.459-467.

[5] Arita T. and R. Suzuki. (2000). *Interactions between learning and evolution: The outstanding strategy generated by the Baldwin effect*. In Proceedings of Artificial Life VII, pp. 196-205, MIT Press.

[6] Bäck T. (1995). *Generalized convergence models for tournament and (μ,λ)–selection*. In L. Eshelman, ed., proceedings of the Sixth International Conference on Genetic Algorithms (ICGA95), San Francisco, CA. Morgan Kaufmann Publishers.

[7] Bäck, T. (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press.

[8] Bäck, T., U. Hammel and H.P. Schwefel. (1997). *Evolutionary Computation: Comments on the History and Current State*. IEEE Transactions on Evolutionary Computation, Vol. 1, No 1, pp.3-17.

[9] Baker J. E. (1987). *Reducing bias and inefficiency in the selection algorithm*. In Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Erlbaum Associates (Hillsdale).

[10] Beasley, D., D.R. Bull and R.R. Martin. (1993). *An Overview Of Genetic Algorithms*. University Computing, Part 1 "Fundamentals", 15(2) 58-69, Part 2 "Research Topics", 15(4) 170-181.

[11] Berg J.O. (1993). *Path and Orientation Accuracy of Industrial Robots*. International Journal of Advanced Manufacturing Technology. 8:pp.29-33, Springer-Verlag London.

[12] Bernhardt, R. and S.L. Albright, editors. (1993). *Robot Calibration*. Chapman & Hall, London.

[13] Blickle, T. and Thiele, L. (1995). *A Comparison of Selection Schemes used in Genetic Algorithms*. TIK Report Nr. 11, ETH Zürich.

[14] Caenen J. and J. Angue. (1990). *Identification of geometric and non geometric parameters of robots*. Proceedings of the IEEE International conference on robotics and automation, pp. 1032-1037.

[15] CAMELOT. (2000). *Ropsim product desription*. Allerød, Denmark.

[16] Chellapilla K. (1997). *Evolving Computer Programs without Subtree Crossover.* IEEE Transactions on Evolutionary Computation, Vol. 1, No. 3, pp. 209-216.

[17] Cramer, N.L. (July 1985) *A Representation for the Adaptive Generation of Simple Sequential Programs*. Proceedings, International Conference on Genetic Algorithms and their Applications, July 1985 [CMU], Pittsburgh, pp183-187.

[18] Davidor, Y. (1991). *Genetic Algorithms and Robotics*, World Scientific , Singapore. ISBN 9-810202172.

[19] DELMIA. *IGRIP Reference Manual*.

[20] De Jong, K.A. (1975). *An analysis of the behavior of a class of genetic adaptive systems*. Doctoral thesis, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor.

[21] De Jong, A. Kenneth and William M. Spears. (1992). *A Formal Analysis of the Role of Multi-Point Crossover in Genetic Algorithms*. In Annals of Mathematics and Artificial Intelligence, Volume 5, 1, 1-26.

[22] Doty, K.L., C. Mechiorri and C. Bonivento. (1993*). A theory of generalise inverses applied to robotics*. International Journal of Robotics Research, 12, pp.1-19.

[23] Driels M.R. and U.S. Pathre. (1994*). Robot Calibration Using an Automatic Theodolite*. Int. Journal of Advanced Manufacturing Technology, 9, pp. 114-125, Springer Verlag London Ltd.

[24] Dror G. Feitelson and Michael Naaman. *Self- Tuning Systems* in IEEE Software March/April 1999, p52- 60

[25] Duelen G. and K. Schröer. (1991). *Robot Calibration – Methods and Results*. Robotics and Computer- Integrated Manufacturing, Vol.8, No.4, pp. 223-231.

[26] Emden-Weinert T., S. Hougardy, B. Kreuter, H.J. Prömel and Angelika Steger. (© 1996). *Einführung in Graphen und Algorithmen* (Introduction to graphs and algorithms) Manuskript, Institut für Informatik, Lehrstuhl für Algorithmen und Komplexität, Humboldt-Universität zu Berlin,
Available at: http://www.informatik.hu-berlin.de/Institut/struktur/algorithmen/ga/.

[27] Everett, L.J., M. Driels and B.W. Mooring. (1987). *Kinematic Modelling for Robot Calibration*, Proceedings of IEEE International Conference of Robotics and Automation, pp. 183-189. IEEE Press.

[28] Everett, L.J. and Tsing-Wong Hsu. 1988. *The Theory of Kinematic Identification for Industrial Robots*. Transactions of the ASME. Journal of Dynamic Systems, Measurement and Control, Vol. 110, 96-100.

[29] Everett L.J. (1993). *Models for Diagnosing Robot Error Sources*, Conference on Robotics and Automation, Vol.2, pp.155-159.

[30] FAMOS®. Carat robotic innovation GmbH. *FAMOS Product description*. Press Release (Flexible Automation 5/98).

[31] Fogel, L.J., Owens, A.J. & Walsh, M.J. (1966). *Artificial Intelligence through Simulated Evolution*. Wiley Publishing, New York.

[32] Fogel, D.B. (1997). *The Advantages of Evolutionary Computation*. In Lundh, D., B. Olsson and A. Narayanan eds., Proceedings of BCEC97: BioComputing and Emergent Computation, Singapore, World Scientific, pp.1-11.

[33] Fonseca C.M. and P.J. Fleming. (1993). *Genetic algorithms for multi-objective optimization: Formulation, discussion and generalization.* In: S. Forrest (ed.), Genetic Algorithms: Proceedings of the Fifth International Conference, Morgan Kaufmann, San Mateo, CA, 141-153.

[34] Fu K.S., R.C. Gonzalez and C.S.G. Lee. (1987). *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw Hill, Singapore.

[35] Goldberg D. E. (1989). Genetic Algorithms in search, optimisations and machine learning. Addison-Wesley.

[36] Goldberg D. E., B. Korb and K. Deb. (1989*). Messy genetic algorithms: Motivation, analysis, and first results*. Complex Systems, 3, pp. 493-530.

[37] Goswami, Ambarish, Arthur Quaid, Michael Peshkin. 1993. *Complete Parameter identification from partial pose measurement.* IEEE International Conference on Robotics and Automation.

[38] Grefenstette J.J. and J.E. Baker. (1989). *How genetic algorithms work: a critical look at implicit parallelism*. In Schaffer, J.D. (Ed.), Proceedings of the Third International Conference on Genetic Algorithms (ICGA3), pp. 20-27. CA. Morgan Kaufmann.

[39] Gruau F. (1995). *Automatic Definition of Modular Neural Networks*. Adaptive Behavior V3N2, pp. 151-183, MIT press.

[40] Harvey I., P. Husbands and D. Cliff: *Issues in evolutionary robotics* in: J.-A. Meyer, H. Roitblat and S. Wilson (eds.), From Animals to Animats 2: Proc. of the Second Intl. Conf. on Simulation of Adaptive Behavior, (SAB92), pp. 364--373. MIT Press/Bradford Books, Cambridge MA, 1993.

[41] Hayati S.A. and M. Mirmirani. 1985. *Improving the absolute positioning accuracy of robot manipulators*. J. Robotic Systems. 2: 397-413.

[42] Holland J.H. (1975) *Adaptation in Natural and Artificial Systems*. Ann Arbor, Michigan: The University of Michigan Press.

[43] Hollerbach, J.M. and C.W. Wampler. (1996*). A taxonomy of kinematic calibration methods*. International Journal of Robotics Research, 14, pp. 573-591.

[44] Hollerbach, J.M. (1998). *Robot calibration lecture notes*. University of Texas at Austin.

[45] Jenkinson, I.D. (2000). *An application of neural networks to improve the accuracy of an industrial robot for offline programming*. Ph.D. Thesis, Liverpool John Moores University, Liverpool, UK.

[46] Jordan, M.I. and D. E. Rumelhart. *Forward models: Supervised learning with a distal teacher*. Cognitive Science, 16, 307-354, 1992.

[47] Judd R.P and A.B. Knasinski. 1990. *A Technique to Calibrate Industrial Robot with Experimental Verification.* IEEE Transactions on Robotics and Automation, Vol.6, No 1, pp. 20-30.

[48] Keane M.A., J.R. Koza and J.P. Rice. (1993). *Finding an impulse response function using genetic programming*. In Proceedings of the 1993 American Control Conference , volume III, pages 2345-2350, Evanston, IL, USA.

[49] Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

[50] Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reuseable Programs*. Cambridge, MA: The MIT Press.

[51] Koza J.R. (1997) *Future work and practical applications of genetic programming*. In T. Baeck, D. B. Fogel, and Z. Michalewicz, editors, Handbook of Evolutionary Computation, pages H1.1--1--6. Oxford University Press.

[52] Ku, K. W. C. and M. W. Mak. (1997). *Exploring the effects of Lamarckian and Baldwinian learning in evolving recurrent neural networks*. In Proceedings of the IEEE International Conference on Evolutionary Computation, pages 617-621.

[53] Luke, Sean. 2000. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.* Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, Maryland.

[54] Ma, D., and J.M. Hollerbach. *Identifying mass parameters for gravity compensation and automatic torque sensor calibration*. Proc. IEEE International Conference. Robotics and Automation, Minneapolis, April 22-28, 1996, pp. 661-666.

[55] Michalewicz Z. (1992) *Genetic Algorithms + Data Structures = Evolutionary Programs*. Artificial Intelligence. Springer Verlag, Berlin.

[56] Miller B.L. and D.E. Goldberg. (1995). *Genetic Algorithms Selection Schemes and the Varying Effects of Noise*. IlliGAL Report No. 95009, University of Illinois.

[57] Ming-Yi Lay. (1994). *Genetic Programming and its application to analyze dynamical systems*. Ph.D. thesis. The University of Texas. Austin.

[58] Montana D.J. (1995). *Strongly Typed Genetic Programming*, Evolutionary Computation, Vol.3, No.2, pp. 199-230.

[59] Mooring, Benjamin W., Zvi S. Roth and M.R. Driels. (1991). Fundamentals of Manipulator Calibration. John Wiley & Sons.

[60] Mühlenbein, H. and D. Schlierkamp-Voosen. (1993). *Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization*. Evolutionary Computation, 1 (1), pp. 25-49.

[61] Nordin P. and W. Banzhaf. (1997). *An On-Line Method to Evolve Behavior and to Control a Miniature Robot in Real Time with Genetic Programming*: Adaptive Behaviour, 5 (2), pp.107–140.

[62] Paul R.P., B. Shimano & G.E. Mayer. (1981). *Kinematic Control Equations for Simple Manipulators*. IEEE Transactions on Systems, Man, and Cybernetics. Vol. SMC-11, No.6, June 1981.

[63] Perkins S. (1998). *Incremental Acquisition of Complex Visual Behaviour using Genetic Programming and Shaping*, PhD thesis, University of Edinburgh, UK, Dec 1998.

[64] Pohlheim H. and P. Marenbach.(1996). *Generation of structured process models using genetic algorithms*. In T. Fogarty, editor, Proc. AISB'96 Workshop on Evolutionary Computing, volume 1143 of Lecture Notes in Computer Science, pages 102-109. Springer-Verlag.

[65] Rechenberg, I. (1973) *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, Stuttgart.

[66] Rosca J.P. (1995). *An Analysis of Hierarchical Genetic Programming*. TR 566, Computer Science Dept., University of Rochester, March 1995.

[67] Roth, Zvi S., Benjamin W. Mooring and Bahram Ravani. (1987*). An Overview of Robot Calibration.* IEEE Journal of Robotics and Automation, Vol. RA-3, No.5, 377-385, October 1887.

[68] Schröer K. (1993) *Theory of kinematic modelling and numerical procedures for robot calibration.* Robot Calibration, eds. R. Bernhadt and S.L. Albright, pp. 157-195, Chapman & Hall, London.

[69] Schwefel, H.-P. (1981) *Numerical Optimization of Computer Models*. John Wiley & Sons, New York.

[70] Shamma J.S. and D.E. Whitney. (1987). *A Method for Inverse Robot Calibration*. Transactions of the ASME Journal of Dynamical Systems, Measurement and Control, Vol.109, pp. 36-43.

[71] Spears, William, M., K. A. De Jong, T. Baeck, D. Fogel, and H. de Garis (1993). *An Overview of Evolutionary Computation*. In Proceedings of the European Conference on Machine Learning, v667, 442-459.

[72] Stäubli: Unimation SA. (1992). *VAL II command reference manual*.

[73] Stone H.W. (1987*). Kinematic Modelling, Identification and Control of Robotic manipulators*. Kluwer Academic Publisher, New York.

[74] Stroustrup, Bjarne. (1997). *The C++ Programming Language (3rd edition)*. Addison Wesley Longman, Reading, MA.

[75] Unimation. (1986). *Equipment Manual 398Z1*, UNIMATE® PUMA® Mark III – VAL™ II Robot, 700 Series, Models 761/762. Unimation Ltd. Telford, UK.

[76] Vincze M., K.M. Filz, H. Gander, J.P. Prenninger and G. Zeichen. (1994). *A Systematic Approach to Model Arbitrary Non Geometric Kinematic Errors.* Advances in Robot Kinematics and Computationed Geometry, 129-138, Kluewer Academic Publishers, Netherlands.

[77] Vincze M., K.M. Filz, H. Gander & J.P. Prenninger. (1996) . *A Systematic and Extendible Model for Arbitrary Axis Configurations.* International Journal of Flexible Automation and Integrated Manufacturing 3(2), pp. 147-164.

[78] Vincze M. J.P Prenninger and H. Gander. (1994) *A laser tracking system to measure position orientation of robot end effectors under motion*. International Journal of Robotics Research, 13, pp. 305-314.

[79] Vincze M., S. Spiess, M. Parotidis and M. Götz. (1999). *Automatic Generation of Non-Redundant and Complete Models for Geometric and Non Geometric Errors of Robots*. International Journal of Modelling and Simulation 19(3), pp. 236-243.

[80] Whitley, D. (1989). *The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best*. In Schaffer, J.D. (Ed.), Proceedings of the Third International Conference on Genetic Algorithms (ICGA3), pp. 116-121. San Mateo, CA. Morgan Kaufmann.

[81] Whitney, D.E., C.A. Lozinski and J.M. Rourke. (1986). *Industrial Robot Forward Calibration Method and Results*. Journal of Dynamic Systems, Measurement, and Control, Vol. 108, pp. 1-8.

[82] Workspace 4, *User Guide Manual*, Robot Simulations Ltd, 1998.

[83] Wu C. (1984). *A kinematic CAD tool for the design and control of a robot manipulator*. International Journal of Robotics Research, 3, pp. 58-67.

[84] Zhuang H., Z.S. Roth and F. Hamano. 1992. *A Complete and Parametrically Continuous Kinematic Model for Robot Manipulators*. IEEE Transactions on Robotics and Automation , Vol.8, No.4.

[85] Zhong X., j. Lewis and F.L. N-Nagy. (1996*) Inverse Robot Calibration Using Artificial Neural Networks*. Engineering Applications of Artificial Intelligence, Vol.9, No.1, pp. 83-93. Elsevier Science Ltd.