Simplified Integration of External Sensors in Industrial Robot Programs

Der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
vorgelegte Abhandlung

D i s s e r t a t i o n

vorgelegt von

Dipl.-Inf. Jan Deiterding

aus Aachen

1. Gutachter:
2. Gutachter:

Tag der Einreichung:
Tag des Kolloquiums:

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The employment of robots in industrial manufacturing has risen constantly in recent years. The robot industry was capable of tripling its revenues in the last ten years to nearly 1.9 billion Euro. About a quarter of all robots installed in German factories is used for welding tasks and nearly 50 percent are operated in car-related industries [52]. Apart from this modern robots are used for packaging, palletizing, handling and assembly tasks. This has made the robot a universal machine that is employed in nearly all areas of modern industrial production such as the chemicals industry, the metal processing industry, mechanical engineering and electronics. However, recent studies predict a saturation of the robot market in the next years, since most large companies are highly automated already [123].

1.1 Motivation

In large companies there is no need for more robots unless there is a significant rise in production. Only if robots are able to solve more complex tasks new areas of application can be attained. The one area where the use of robots may be enforced and is likely to increase over the years is in *small and medium enterprises (SME)*. Unlike large companies, SMEs usually produce goods in much lower quantities and have a higher turnover rate in production. However, robots are only rarely employed here. This is caused by two factors: Firstly, robots are expensive. So there must be a significant advantage in using a robot instead of a human worker in terms of production time and cost. This is closely related to the second factor. Robot programming is a difficult task demanding skilled experts. While big companies can afford these experts due to a high number of robots, SME usually cannot, because they own only one or two. In order to make robots more attractive for SME, they must be able to act flexibly and be programmable by non-experts. Then the ability to compete will rise. The aim of this work is to look into this in more detail.

Figure 1.1: Industrial robots. Left: Staeubli RX130. Right: Kuka LBR.

1.1.1 Use of Industrial Robots

The advantages of a robot in comparison to a human worker are its high precision, higher speed of execution and the fact that robots can execute tasks without any signs of exhaustion. For a robot program to execute a given task mainly consists of a series of commands for movements and grasping / releasing an object or controlling some kind of tool mounted to the robots wrist[1]. Control structures such as if- or while-commands are rarely employed. This allows for a very simple way of programming. One approach is to guide the robot through the task and record all motions, so-called *programming by demonstration* (PbD). In production, this collection of motions is then repeated over and over, usually at significantly higher speed. This and other ways of robot programming are more closely analyzed in Chapter 2.2.
To give an example, two industrial robots are shown in Figure 1.1. The RX130 from Staeubli (left) is used in industrial contexts for several years by now. The LBR from Kuka (right) is a new design by the German Aerospace Center, geared towards small and medium enterprises.

1.1.2 Sensor Integration

Industrial robots lack some characteristics that are relevant for more sophisticated tasks. They have no means to interact with the environment they are working in. In order to execute a task, one must ensure that all parts are always arranged in precisely the same position and orientation. In addition, part variations, e.g. in terms of size, present enormous problems to robots. Movements must be adapted to compensate the variation.
There are two ways to deal with these imperfections: Firstly, mechanical compliances as well as damping and feeding devices are created to ensure correct grasping and handling with a fixed set of motions. This approach yields the advantage that no perception capabilities

[1]There are a lot more commands in modern robot programming languages. These include commands for input/output and error handling. But all of these provide additional features, but are not mandatory.

are required of the robot. But the construction of fixtures and mechanisms is elaborate both in time and cost. Additionally this construction requires expertise in that field of work. Another disadvantage is that these fixtures can only rarely be re-used when the task changes in any aspect or is abandoned. So this approach is only advisable if the task will be executed repeatedly for such a long time that the time required to setup the workspace is neglectible. If the task is executed for only a short period of time in relation to the time required for setup, there is no sense in employing a robot at all, because the overall costs will be too high to justify its use.
The second approach is to equip the robot with sensor capabilities. *External sensors* are installed in the robot and in its workspace to measure changes in the environment. In this case a robot motion is not fixed but altered online during task execution to reflect the information gained by the sensors. With this approach a robot is truely flexible in the sense that it can react to possible changes in its workspace provided the sensor is capable of detecting it. Additionally, sensors can be re-used for different tasks. But there is a significant downside: The information inherent in the sensor signal must be extracted and analyzed in order to alter the movements of the robot correctly. This yields to a series of problems:

- Sensor data processing is a very complex task. It is relatively straightforward to interpret the data of by a distance sensor. This gives us one-dimensional information about the distance between itself and the next obstacle in its range. But it is a much more difficult task to analyze an image provided by a camera to detect an object. So the developer[2] must possess substantial knowledge in sensor data processing to evaluate the sensor signal and extract the relevant information.

- A robot is a real-time system moving in and altering its environment at high speed. Because of this the sensor signal must be processed fast enough to allow the robot to react in time to a change. Otherwise motions based on external sensor data must be slowed prolonging execution time.

- Sensor data processing must be integrated into the robot program. Now there are not only robot commands but sensor commands as well. Because the robot is moving and may occlude objects from the sensors, these two tasks must be adjusted accordingly to ensure correct and safe execution.

These three factors are the main reason why external sensors in industrial manufacturing are only used where absolutely necessary. Skilled experts are required with knowledge in both areas, robot programming and sensor data processing. Typical examples of sensors used in industrial contexts are shown in Figure 1.2.
In large companies a robot shall execute a task up to a hundred thousand times or even more. In this case, a mechanical solution is preferable in terms of production time and cost. But there is a limit to the range of applications such robots can deal with. For example

[2]The use of the male pronoun 'he' in context with the developer is purely practical and does not represent reality.

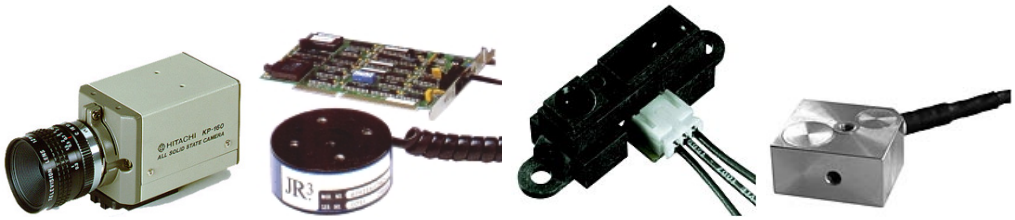


Figure 1.2: Different types of sensors typically used in industrial manipulation tasks. From left to right: Imaging sensor, force/torque sensor, optical distance sensor, vibration sensor.

if flexible or organic materials are to be handled, external sensors are mandatory, because these materials differ in size and shape in every execution.
So one key problem is to find ways to allow for a programming of flexible robots by non-experts. The target group are SME because of they usually fewer experts and have a high turnover rate requiring re-programming at regular intervals. It is more important to create the program quickly than to optimize it with regard to its execution time. This is contrary to the needs of large companies like car manufacturers. Here, execution time is critical because even small increases will add up significantly. In order to minimize execution time a lot of consideration goes into laying out the workspace and designing the robot program in large companies. This can take up to several weeks. Unlike large companies in SME the task of creating a new program for a new task must be achieved as quickly as possible. Nonetheless large companies are interested in flexible robots that are easy to program as well. One example for this is the American car manufacturer General Motors. The head of the R&D department, Robert Tilove, has emphasized General Motor's interest in this area of research [119]. Focussing on SMEs, special projects have been founded to develop new solutions for easy robot integration [9] and this topic is discussed intensively at various scientific conferences [54], [5], [98], [35].
For the same reasons the employment of fixtures and feeding devices is not an option for SMEs. The time and cost required to create and install such devices is not justified by the lot size. External sensors must be used instead.
There are many approaches to create static robot programs easily. These are described in detail in Chapter 2. However, it is a much more difficult task to extend a static robot program to react flexibly due to the three problems described above.
So, a key issue is how external sensors can be integrated easily into a robot program by persons with only basic knowledge in robot programming and sensor data processing.

1.1.3 Adaptivity

For SME, execution time is not that crucial, but it is still an important factor. The more often a task is repeated, the more important execution time becomes. While a task must always be executed correctly, the more often the program is executed, the more important the actual time required to complete the task gets. If the task is executed only a couple of

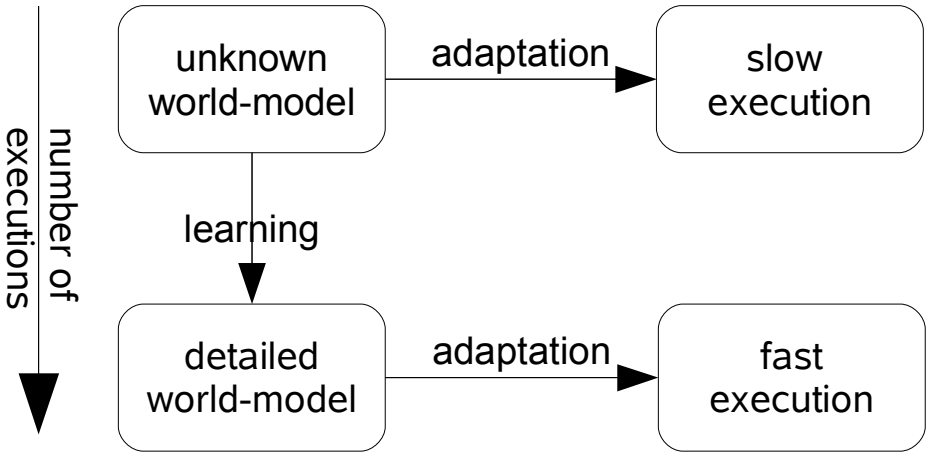


Figure 1.3: Adaptive robots re-use knowledge gained in previous executions to optimize the execution with respect to the execution time.

times, it does not matter so much, as long as the execution time is still acceptable. But, since every increase in execution time must be multiplied by the number of repetitions, this will amount to a significant delay for higher numbers of repetition.
Even if the robot can be easily programmed to accomplish a task, a human worker is more suited, if he can accomplish the task more quickly than the robot. As outlined in the previous section, optimizing a robot program is a time-consuming task and if external sensors are employed, this becomes very complicated. So another characteristic a robot should possess is *adaptivity*.
Adaptivity in an industrial robot context is the capability to re-use knowledge gained in previous executions of the same or a similar task to optimize the current task in terms of execution time or robustness. This definition is made in analogy to a human worker. A human will be slow and make a lot of mistakes when he is shown a new task and executes it for the very first time. But the more often he repeats that task, the more experience he will gain and optimize its movements as well as recognize critical situations that may surmount to an error. If the robot possesses adaptive characteristics, it will also learn to optimize the execution of a given task based on knowledge it has gained in previous executions (see Figure 1.3).
The difference between flexibility and adaptivity is that flexibility concerns the robot's capabilities to compensate a change in the workspace in the current execution of the task. Any knowledge gained in this execution is forgotten when the task is repeated. Adaptivity describes the capability to store and re-use this knowledge in another execution for a more precise or faster execution.
The other key issue is if and how a robot can learn to optimize its movements to ensure an acceptable execution along the criteria of correctness and execution time. If this is possible, parts of the difficulty of sensor data processing can be passed from the programmer to the robot. This further reduces the complexity of programming and the development time because now only the basic parameters and instructions have to be outlined by the programmer [18], [73], [74].

1.2 Goals

The A-Bot project was formed in 2006 to find solutions for the issues of easy sensor integration and adaptivity concepts to decrease execution time. A-Bot is short for *adaptive robot programming*. The focus of A-Bot is not to find new ways of creating robot programs but enabling persons with only basic expertise in robot programming and sensor data processing to extend a given static robot programs with external sensors.
The resulting program shall be *flexible* in terms of dealing with object variations such as search motions, insertion procedures and object drifts as well as being *adaptive* in terms of self-optimization of program parts to reduce execution time.
A static robot program shall be given. This creates no limitation because today there are various ways to create such programs with minimal knowledge in robot programming. Some of these approaches will be discussed in detail in Section 2.2. As stressed before the problem lies in the task to integrate external sensors. While there are approaches to a unified development process yielding flexible or adaptive robot programs straight away, these are either geared towards specific applications or aimed at experts. We will discuss these approaches in Sections 2.3 and 2.4.
When creating the basic program and integrating sensors are regarded as separate tasks, different approaches can be taken to execute each phase. For example, designing the static program can be achieved using textual programming or programming by demonstration techniques. The approach taken does not influence the second phase when external sensors are incorporated. Because of this, we propose a modification of the classical approach to robot programming, that employs three distinct steps, namely:

1. A programming step in which the program is designed offline
2. A testing and modification step that is performed online
3. The execution of the program in production mode

In our approach some additions are introduced, that interweave with each other (see Figure 1.4):

1. In the first phase, the general program flow and the basic positions of objects, trajectories, etc. are laid out. This step is identical to the first step of the classical approach and can be achieved using the concepts mentioned in the Sections 2.3 and 2.4. Note that no external sensors are used up to now.
2. In the second phase, the developer can add flexibility to the program flow. External sensors are integrated into the program and mechanisms are created to deal with changes requiring flexibility of the robot. This task is performed both offline and online.
3. Shortly after the introduction of external sensors, the testing and validation of the program begins.

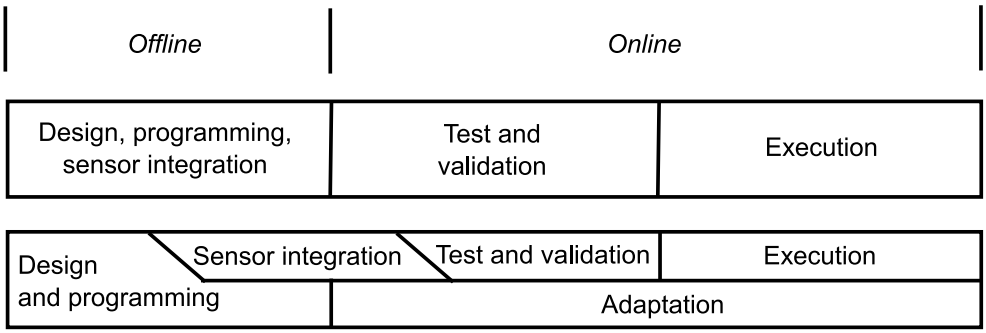


Figure 1.4: Comparison of the classic approach to robot programming (top) and the interleaving approach presented here (bottom).

4. As soon as the program is run for the very first time, automatic adaptivity methods are employed to re-use knowledge in order to optimize the program. As long as the program is tested by the developer, this is done in a supervised way.

5. After testing, the program is executed in production mode. Adaptation may continue in an unsupervised way.

We assume that the developer possesses basic knowledge of robot programming. This means that he knows the most frequently used commands such as movement-, speed- and grasp/release-commands. Knowledge of sensor data processing algorithms shall not be required but a basic understanding of the information contained in the sensor signal is necessary.
A-Bot focusses on *industrial handling tasks*. These tasks deal with grasping and releasing objects as well as manipulating them (e.g. insertion, stacking, palletizing, etc.). Some examples for these kinds of tasks are shown in Figure 1.5. These tasks are *position-centered* in the sense that the robot must know where the objects are placed, what their orientation is and in which way they can be manipulated. It is of minor importance in which way these objects are moved along the trajectories. A typical task is illustrated and described in Section 3.1.1. A more detailed description of these kind of tasks is given in Section 3.3. Explicitly not included are processing tasks, such as welding and painting of objects. These tasks are *trajectory-centered* in the sense that the robot must follow a trajectory or the contour of an object as exactly as possible, e.g. for spot-welding purposes. These tasks require a different approach and already are covered by various research groups, e.g. [28], [36], [86].

1.3 Problem

This work is meant to outline a first approach to solve (parts of) the goals outlined in the previous section. Based on the universal requirements, the purpose of this work is to find solutions to the following problems:

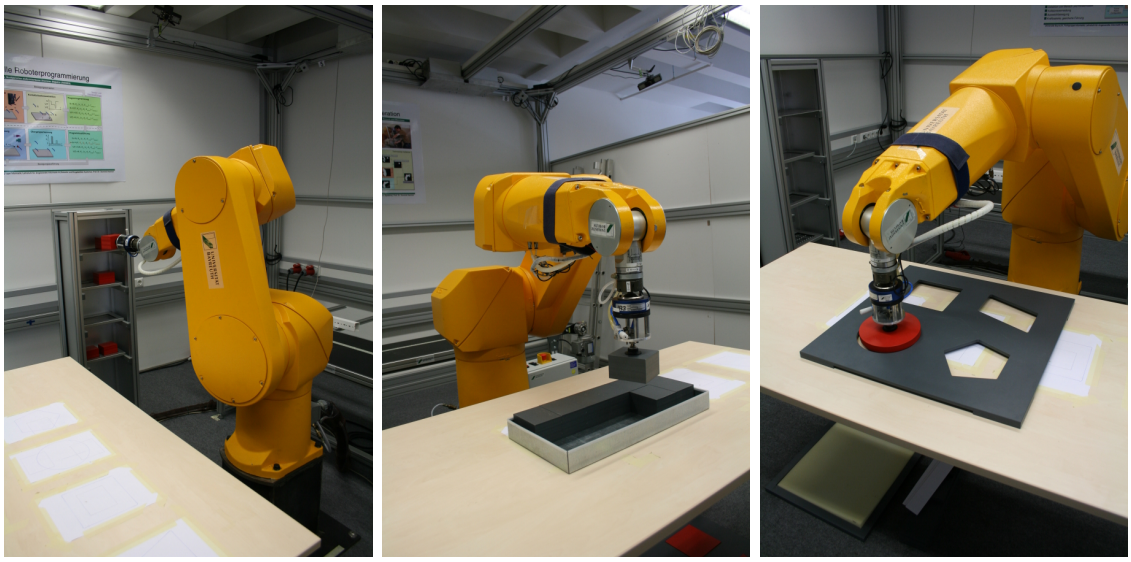


Figure 1.5: Examples for robot handling tasks. Left: *Stacking* objects into containers. Middle: *Palettizing* objects into boxes. Right: *Inserting* objects into corresponding slots.

- Analyzing the general use of external sensors in robot manipulation tasks
- Analyzing flexibility and adaptivity measures for robot manipulation and developing concepts enabling a robot to react to changes in the workspace
- Developing a first implementation of a universal programming concept to integrate external sensors into a static robot programm

The resulting concepts shall adhere to the following criteria:

- The complexity to integrate a given sensor into a robot program.
- The expertise a developer needs to create a flexible and adaptive robot program.
- The universality of the approach taken.

The implementation does not need to be exhaustive. On the contrary, there shall be room for further expansions, e.g. solutions tailored to specific task domains. In this work, a first development cycle shall be laid out. With this cycle, the developer shall be able to analyze the problem of workspace variations, choose suitable sensors for compensation and integrate the sensor in a pre-determined way into a given static robot program. We are looking for a first solution to each of the aspects outlined above. There may be other possible solutions as well, so future enhancements of the implementation must be taken into account as well.

1.4 Delimitation

The purpose of this work is to develop a programming concept to integrate external sensors into a given static robot program for position-centered tasks, such as handling. No part of this work treats trajectory-centered tasks. In these tasks, the robot shall not form any decisions, but follow a given trajectory at a set speed and allowed deviation. There is no contact between the robot and its environment. Handling tasks differ in the sense that here objects are brought into contact with each other. Additionally, these objects may differ in size, position and orientation. Because of the different foundation of these tasks (trajectories instead of positions) other concepts have to be applied here. Projects dealing with trajectories have already been mentioned in section 1.2.
Neither will we present detailed solutions for specific kinds of manipulation tasks, e.g. *peg-in-hole* tasks. We will try to formulate a general approach that can be used for all position-centered robot tasks. All solutions developed in this work shall be applicable to all kinds of tasks. However we will try to allow for an integration of additional, more application-focused extensions.
With respect to the type of robot used, the concepts and solutions realized in this work will be kept as general as possible and shall not be aimed towards a specific type of robot or robot manufacturer. It should be possible to realize the concept developed in this work on every robot system capable of grasping objects. This includes the avoidance of *domain specific languages* (DSL). Such an approach would severely limit the field of application to specific tasks and complicate a transfer to another domain. Because of this, we will outline the concepts in general robot commands. It a next step, the transfer of these concepts to a DSL can be attempted. But this is not part of this work.
The above also applies to the type of sensors employed. We will not deal with the problem of physically integrating a sensor into a robot system, that is, installing the sensor somewhere in the workspace and ensuring that the sensor signal is accessible to the robot system. Given modern connecting systems and interfaces such as USB, this problem lies more in the domain of electrical engineering than computer science.

1.5 Overview

This work is organized as follows: In Chapter 2 we give an overview of other approaches to sensor based robot programming and outline the general difficulties of this kind of programming and shortcomings of other approaches thus legitimating our approach. In Chapter 3 we describe how sensors can be used to monitor workspace changes. We will classify these changes and describe ways to transform sensor data into an abstract description of the change at hand. Additionally, we will describe a method to compute this transformation function during task execution. In Chapter 4 we analyze the use of external sensors for robot manipulation tasks and develop a general programming concept to integrate such sensors into a robot program easily. In Chapter 5 we focus on workspace changes that require search motions. We show how adaptivity methods can be employed to automatically

optimize these searches. In Chapter 6 we deal with the drift of objects that are an undesired factor occurring in manipulation tasks. We explain how objects can be monitored for a drift and describe ways of adapting to and correcting a drift automatically. In Chapter 7 we describe a prototype implementation of the concepts developed in Chapters 3 to 6 and show how this prototype enables developers to create flexible robot programs with adaptive capabilities. Chapter 8 summarizes our work and gives an outlook on future research.

Chapter 2

State of the Art

In this chapter we give an overview of existing approaches to flexible robot programming. We describe various approaches to robot programming and evaluate them with respect to their usability and the suitable range of applications. Based on this we conclude why there is a need for a general programming concept for industrial handling tasks requiring external sensors.
The rest of this chapter is organized as follows: In section 2.1 we describe a way to classify robot programming. We rate programming concepts according to their usability for non-experts. In section 2.2, 2.3 and 2.4 we discuss various styles of robot programming. In section 2.5 we conclude why there is a need for an easy-to-program adaptive robot system. We will discuss other works that are related to the design of certain aspects of the whole system in the immediate chapter concerned.

2.1 Overview

A good overview of the current state of industrial robot programming is given in [110]. It acknowledges that "the use of robots in small and medium-sized manufacturing is still tiny". One of the reasons given is the problem of human-friendly task specification.
A review of different robot programming systems was conducted by Lozano-Perez in 1983 [82]. Systems were classified into three classes: Guiding systems, robot-level programming and task-level programming. In guiding systems (or *direct programming*), the developer moves the robot physically through every position of the task and records all operations. In execution, this recording is then simply played back at higher speed. Robot-level programming systems (or *explicit programming*) provide the developer with a special robot programming language to describe the task (usually in a textual way in an editor). In task-level programming (or *implicit programming*), the developer only describes the tasks or goals to be achieved. The actual program is then created automatically from this description. Guiding systems are a way of direct programming occupying the workspace so no other tasks can be executed while designing the new robot program. Robot- and task-level programming are performed offline using either explicit (robot) or implicit (task) comman-

	Direct	Explicit	Implicit
Static	Teaching	Manufacturer languages	Montage plans
Flexible	Programming by demonstration	Action-primitives	Contact based execution
Adaptive	Programming by demonstration	Learning skills	Learning skills

Table 2.1: Examples for different types of programming for industrial robots classified by type of programming and desired capabilities of the robot.

ds.
The work of Lozano-Perez was extended by Biggs in [20]. Biggs introduces a second scheme: Automatic programming, manual programming and software architectures. Guiding systems and robot-level programming are classified as manual programming, while task-level programming is classified as automatic programming. Software architectures "are important to all programming systems, as they provide the underlying support, such as communication, as well as access to the robots themselves." [20].
In our work we will use the classification given by Lozano-Perez and extend it by an additional classification: Is the robot program static, flexible or adaptive? A static robot program does not react to its environment and repeats the same task with no deviations over and over. Flexible robot programs are able to react to their environment and alter their movements should the need arise. But this information is "forgotten" in the next execution. Adaptive robot programs re-use information gained in previous executions to further optimize their correctness and speed of execution. Thus adaptive programs form a subclass of all flexible programs. External sensors are required for flexible and adaptive programs, whereas a static robot program does not need them.
Table 2.1 gives an overview of different robot programming concepts with one example each. These concepts and others are explained in detail in the following sections.

2.2 Static Robot Programming

Static robot programming without external sensors is de-facto standard in industrial applications. The avoidance of external sensors allows for very fast robot programs and fixed task cycles. In large companies robots are mainly programmed either directly or explicitly.

2.2.1 Direct Approaches

Direct approaches are moving the robot by using a handheld-device and manually adjusting every joint until the desired position is reached. There are also more intuitive methods using a 6D mouse or moving the real robot by the use of force sensors and zero-force control

[126]. The main advantage of direct programming is that task specific knowledge does not have to be encoded in a textual form.
This *walk-through* or *playback* approach seems to be convenient when robots are to be used in SMEs [108]. A current utilization of this approach is the braiding of carbon fibres [121]. Here the robot moves a form through a radial braiding machine. The robot can be adjusted to a new form by the workers without any knowledge of robot programming. In order to do this, the worker moves the form by pushing and pulling it through the machine to achieve an optimal result.
Hollmann [59] and Pires [98] have developed systems that add speech recognition to this approach. The worker not only moves the robot but also uses verbal commands to instruct the robot to use different types of motion. In a second step a graphical system is used to post process the task before it is run in execution mode.
Soller [68] uses multiple demonstrations of the same task to recognize counting-loops and to generate a correct trajectory from digressive examples.

2.2.2 Explicit Approaches

Explicit programming is mainly done the use of system specific languages provided by the manufacturer. Most manufacturers use their own type of language that may differ significantly from the other, making the porting of an existing task to a robot from a different manufacturer hardly possible. Exemplarily for this concept is the VAL3 language developed by Staeubli [10]. A major problem is that the commands of these languages are focused on moving the robot, which makes them hard to understand for non-experts because they are not task specific. Microsoft has taken the effort to create a framework for a universal robot programming language for industrial as well as mobile robots [7] but in industrial settings this language is practically never used.
Graphical (or icon based) programming systems provide an alternative to text based methods for manual programming. They require manual input to specify actions and program flow. Graphical systems typically use a graph, flow-chart or diagram view of the robot system. One advantage of graphical systems is their easy usability, which is achieved at the cost of text based programmings flexibility. One example is the icon-based system Lego Mindstorms NXT [11] that aims at the home consumer using Lego bricks to create robots (see Figure 2.1). Bischoff et al. [21] have produced a prototype style guide for defining the icons in a flow chart system based on an emerging ISO standard.

2.2.3 Implicit Approaches

Implicit programming approaches for static robot programming are very difficult, due to the fact that imprecisions arise when an implicit command is translated into a series of robot motions. These can only be resolved by using external sensors. *Montage plans* can be used to extract a series of assembly motions from a (graphical) construction manual or CAD data describing the assembly [115]. This can be done if the dimensions of all parts are constant and there are sufficient tolerances.

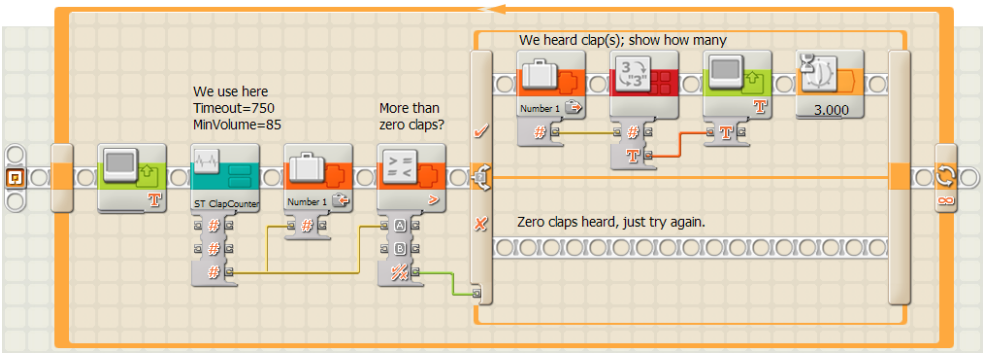


Figure 2.1: Icon based programming using the Lego Mindstorms programming language. Image courtesy of http://www.tau.ac.il/stoledo/lego/ClapCounter/

A graphic representation of the workspace is created in *virtual programming*. The developer moves a simulated robot in a virtual workspace whereby all actions are recorded. This recording is later translated into a textual robot program in the manufacturers language and executed [66]. The advantage of this approach is that the robot remains free to perform other tasks, but as with montage plans, the tolerances must be sufficient. Additionally, modelling the workspace and creating an exact robot model is time consuming. There is a commercial solution called *Robot Studio* by the manufacturer ABB [1] (see Figure 2.2). CAD models can be loaded into the simulation to ensure an exact modelling of the workspace. Already integrated into the software are CAD models of all robots manufactured by ABB.
In summary, there are existing approaches to ease static robot programming. But for SME, the robot must also be able to act flexibly. In order to speed up the set-up of the task, adaptive capabilities are required as well, so none of the approaches above can be taken.

2.3 Flexible Robot Programming

Flexible robots possess the advantage that they can cope with material part imprecisions and deviations with regard to their dimension, position and orientation. This is a key factor for SMEs, but is also of interest for large companies. Unlike static robot programs, the developer must not only specify the correct movements of the robot but also integrate sensor data processing. This further complicates the development.

2.3.1 Direct Approaches

Programming by demonstration is a method where a human gives a demonstration how the task shall be executed. The robot does not have to be used during demonstration (see Figure 2.3). Instead the demonstration is recorded using (multiple) cameras as well as other sensors such as data gloves, microphones, etc. A robot program is generated from this recording. Usually the task is composed using *skill libraries*, encapsulating a fixed se-

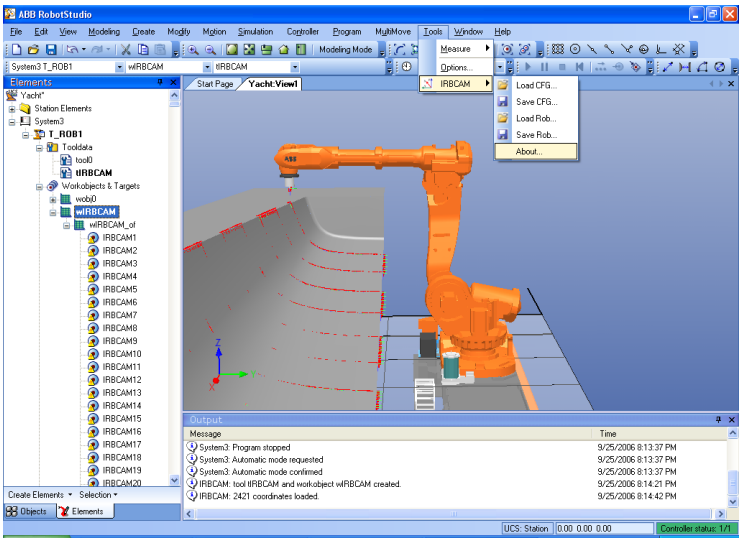


Figure 2.2: Screenshot of the robot studio software by ABB. Image courtesy of http://www.irbcam.com/rev.asp

ries of robot motions into an atomic operation. A skill library is usually developed for a specific domain encompassing the most common operations of this domain. This approach yields the following disadvantages for the creation of flexible robot programs: Firstly, the task usually must be demonstrated more than once. This is because the program generator must be able to distinguish between task-relevant and insignificant characteristics. Secondly, skill libraries are limited to certain domains and are constructed by experts. This is a contradiction to the requirement of a universal applicability. An implementation of this approach is described in [116]. In the works of [13], [103] and [30] statistical methods are used to extract the relevant information of multiple demonstrations of a task. The resulting program is generalized to a certain extent. These works distinguish between relative and absolute robot motions. A detailed survey about programming by demonstration is given in chapter 59 of [110].

All direct approaches presented so far have in common that they only allow for the programming of sequential programs. XProbe [113] is a programming environment that uses a hybrid approach between direct and explicit programming. The user is guided through the programming process by dialogs. In the dialogs, task frames and subroutines are defined. Robot motions are defined by moving the robot and functions can be called using a set of buttons. This allows for more complex behaviour of the robot than a strictly direct approach. In addition sensor based decisions can be integrated as well. The user must activate the sensor with a special command and then execute the motion. As soon as the sensor triggers a set condition, the robot stops and the motion is saved. Afterwards the developer must specify which subroutine should be called if this condition is not met throughout the whole motion. Complex sensor based operations such as insertion strategies cannot be realized with this approach.

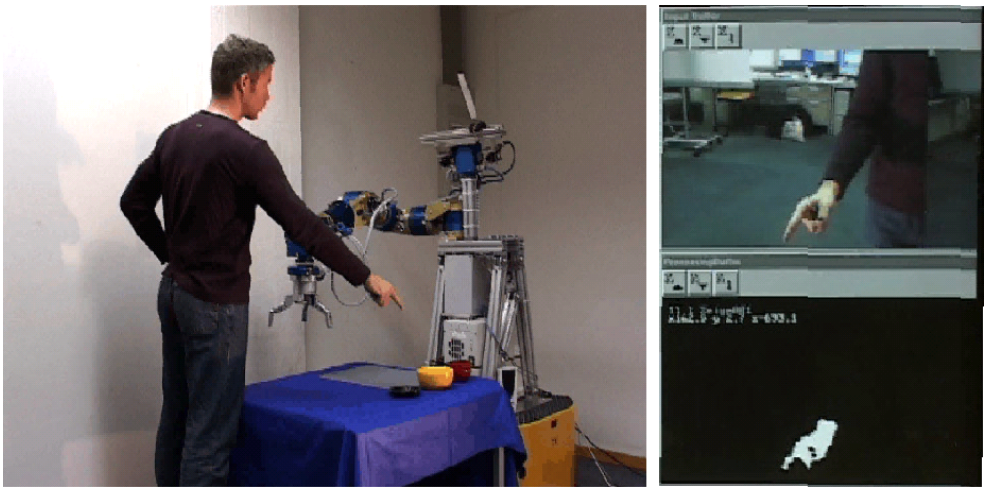


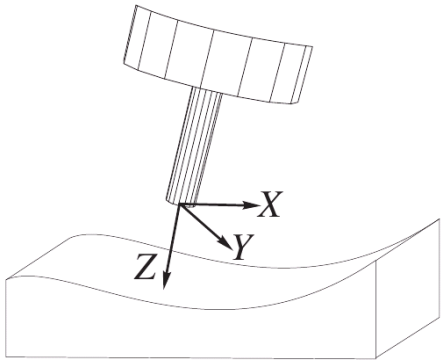
Figure 2.3: Programming by demonstration. Image courtesy of [100]

2.3.2 Explicit Approaches

Robot manufacturers are beginning to recognize the significance of easy sensor integration. Kuka [6] is offering a new robot system with a robot language called *RSI* with integrated sensor data processing capabilities. But so far only one-dimensional sensor data may be processed and the developer must specify exact functions and connections to work with this data. Most sensor manufacturers offer libraries for easy access to the sensor data in the robot programming language, e.g. Adept [2] for the language V+ by Staeubli. But again, the developer must have a very precise understanding of the information contained in the sensor signal.
Scientific approaches to ease flexible robot programming are *functional robot programming languages* [96], [62]. These approaches have not yet been picked up by commercial manufacturers. One reason for this is that the developer must think in functions rather than declarations, which requires a whole new way of programming.
There are a variety of conceptional works to solve specific tasks using sensors. A comprehensive coverage would exceed the scope of this work, so we will only give a short survey of one 'classic' problem of robot assembly: In a *peg-in-hole* task the robot is faced with the problem of inserting an object in a similarly shaped hole. The tolerances are low and the inserting procedure is not trivial. Various solutions exist using different kinds of sensors as well as robots [33], [112], [78], [17], [31], [65], [129]. But all of these works only investigate if there is a solution for a given instance of the problem. There is in no way any concern if these solutions can be re-implemented by non-expert developers. One approach to overcome this, is to encapsulate the solution into a skill, that can be re-used as an atomic operation in a robot program, transforming the solution to an implicit programming approach. Even then the solution is limited to specific instances unless it can be parametrized on a very broad scale (which raises the difficulty of easy application). One example of this kind of skills is given in the next section.

2.3.3 Implicit Approaches

Right on the border between explicit and implicit programming are *action primitives*. These are a form of skills created to be highly independent from the type of robot and sensor used (although there are some limitations). An action primitive is a universal concept to modify a robot motion directly by sensor signals. Sensor data processing is integrated directly into a primitive. For example, a motion can be altered or stopped based on the current sensor signal. Primitives originally were developed to describe complex operations as a sequence of these primitives without having to use exact robot commands. Implicit programming can use primitives to describe some aspect of the task as a skill. But there are some disadvantages to this approach: Firstly, it is not possible to use case distinctions or repetitions in an automated way. This must be done by the developer. Secondly, programming in primitives is by no means intuitive or easy for non-experts. This is because the sensor conditions within a primitive work on the raw sensor data. So the developer must have a very precise understanding how the sensor signal will change during execution. In addition there is a large number of parameters for each primitive that must be specified by the developer. This approach realizes a universal concept but is too complex for non-expert developers. Works using action primitives are [49], [75], [88], [89], [116].
The foundation of action primitives is the *task frame formalism* (TFF) introduced by Mason [86] in 1981. The main idea is to describe robot motions not only in Cartesian coordinates but also in terms of desired force/torque measurements for specific coordinates and orientations. The robot then alters the trajectory accordingly to the measurements. This concept, and variations of it, is widely used in modern robot programming [38], [49], [76], but Bruyninckx and De Schutter note in [28], that "it cannot cope with all possible constrained motion tasks". There are other drawbacks as well: Firstly, this concept is geared towards trajectories only. Localization and classification tasks are virtually impossible. Secondly, while [86] claims, that the concept "serves as a simple interface between the manipulator and the programmer", this is only true for developers with reasonable experience in robot programming. It takes some serious practise to define correct and sensible task frames for tasks outside the demonstrational domain. Kroeger mentions, that "for non-advanced program developers it is highly demanding to utilize all available functions in an optimal way." [76].
Using a *contact state* based approach, a force/torque sensor is used to connect parts. No explicit movement commands are given, but only the contact states of the parts involved. The robot program is then described as a contact state transition graph. Recognizing the different states and computing the next movement is performed by the robot during execution. This approach is mainly used when assembling elastic materials such as wires. The advantage is that the developer does not need to think about the trajectory of the robot but only has to specify the contact state changes. But for complex assemblies these states can only be computed using automated approaches, which again requires a detailed simulated model of the workspace and the physical properties of all parts involved. Virtual programming systems have been used to extract these states [69] (see Figure 2.5). The main problem with this approach is the difficult sensory supervision of contact state



move compliantly {
with task frame directions
xt: velocity 0 mm/sec
yt: velocity 0 mm/sec
zt: velocity v mm/sec
axt: velocity 0 rad/sec
ayt: velocity 0 rad/sec
azt: velocity 0 rad/sec
} until zt force < -f N

Figure 2.4: Specification of a force controlled trajectory using the task frame formalism. Image courtesy of [28]

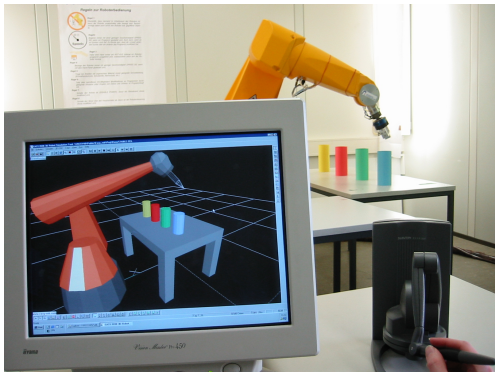


Figure 2.5: Virtual programming. Image courtesy of [69]

changes [57], [127], [128]. It is also possible to use other sensors such as cameras [14].
To alleviate programming a variety of approaches were developed focussing on SMEs. But these solutions are once more geared towards specific tasks and are often based on physical modifications of tools and workspaces [9], [25], [81], [98], [122]. These approaches can not be transferred to a universal approach to robot programming.

2.4 Adaptive Robot Programming

Adaptive robots differ from flexible robots in that they can also store information for re-use in subsequent executions. While in theory external sensors are not necessary for this task[1], their use is common. All adaptive robot programs are flexible as well, so the distinction to flexible robot programming is made solely by the fact if knowledge is re-used or not.

[1]One could simply measure the execution time of the task and use genetic algorithms to optimize the trajectories.

2.4.1 Direct Approaches

Usually a developer must demonstrate the same task repeatedly if programming by demonstration approaches are used. One could argue that this is some kind of adaption. Nevertheless we have chosen to classify this approach as flexible but not adaptive programming. The reason for this is that the robot does not infer information on its own but the resulting program is generated using an existing set of demonstrations.
There are approaches to teach a robot skill libraries [44], [94], but these are aimed at creating a library of skills for a later use by a developer.
Another approach is made by Dixon [45]. A decrease in programming time is accomplished by predicting waypoints in future robot programs and automatically moving the end-effector to the predicted position. Positions from executions of other tasks are stored in a database to achieve this. When the developer teaches the robot a new task, a comparison is made to extract similar positions from the database. It is then inferred if the robot can perform a predictive motion to reduce teaching time. But again, no adaptation can be carried out during the actual task execution.

2.4.2 Explicit Approaches

As was the case with flexible robot programming, in principle all manufacturers' languages allow for the manual programming of adaptive traits. But this requires expert knowledge not only in robot programming and sensor data processing but usually also reasonable experience in machine learning.
The works of Dauster [36] and Bicker [19] deal with adaptive controlled movements along surfaces. Dauster uses a classical proportional/integrated/derivative (PID) controller, whereas Bicker uses a fuzzy-rule control scheme. In both works controlled movements are evaluated in terms of speed and average error and the parameters are optimized with regard to the next execution. While these solutions can be encapsulated into single commands, they are of no use for manipulation task where the focus is on positions instead of trajectories.
Simon [111] has proposed a strategy where the robot program incorporates control primitives with adjustable parameters and an associated cost function. A search algorithm uses experimentally measured performance data to adjust the parameters to seek optimal performance and track system variation. Unfortunately, this research has not been continued.
In principle, there are works to find an optimal solution for a parametrized task. For a long time this was done offline [85], [90], but with increasing computational performance nowadays this can be performed online as well [107], [120]. While this allows to adapt a given task to its surroundings, it is not universally applicable. Once more, there is the problem that a complex series of motions and calculations is encapsulated into a single skill. Another disadvantage is that the optimization algorithms themselves are not universal but tuned to a specific task.
Thrun has developed a programming language extension of C++, called CES [118], specifically targeted towards mobile robot control, with the goal of facilitating the development of such probabilistic software in future robot applications. CES extends C++ by imple-

menting two ideas: Firstly, computing with probability distributions, and secondly, has built-in mechanisms for learning from examples as a new means of programming. The author claims that CES "may reduce the code development by two orders of magnitude". For our work this is not usable as the focus of his work is on mobile robots. Furthermore his extension is not aimed at non-experts.

2.4.3 Implicit Approaches

The development of adaptive implicit robot programming systems is linked very tightly to programming by demonstration approaches. A skill is developed not only to present a flexible solution to a given general task description, but also to be able to optimize itself across multiple executions [94]. Whereas the skill shall be as general as possible, e.g. "insert a peg into a hole", the subsequent adaptation step is geared towards narrowing this general solution to the very specific instance given in the task description. Reinforcement learning strategies and artificial neural networks have been successfully used to achieve this [55], [83]. But again, these solutions just show general feasibility but cannot be transferred to other systems or tasks by non-experts.

2.5 Conclusions

Following the argumentation of Chapter 1.2, we can see that static robot programming is not an option in order to introduce industrial robots to SMEs. For this customer group robots must at least be able to react flexibly to their surroundings. Adaptive traits are even more preferable in order to automatically reduce execution time.
In the domain of flexible programming a large part of the existing work is dedicated to proofs of feasibility for concrete tasks with explicit solutions and specific sensors. There are only a couple of universal approaches abstracting from a given type of sensor. But even those are geared towards experienced programmers, who have gained a precise understanding of the problems of a given task.
Even though there are projects for a more intuitive kind of programming, these are either limited to certain task areas (skills) or impose high requirements on the developer's experience. Both factors are hindering to a general concept for robot programming for non-expert developers.
Summarizing the works discussed in the previous sections, we can say that there are various approaches to allow non-experts to develop robot programs, but most of them focus on static robot programs. If the robot shall be capable of integrating sensor signals during task execution, there are only a couple of projects that have a broader focus than mere feasibility studies. These concepts focus on either intuitive or fast robot programming, but not both. Furthermore, none of these concepts can be transferred to the other criteria respectively. Either the intuitively generated programs are very difficult to optimize or programming is very complex (e.g. caused by a large amount of parameters), so that execution time is satisfactory but the development is practically impossible for non-experts.

Based on this analysis we come to the following conclusion: The only way to get to a system, that can be used universally to program manipulation tasks, is to use an explicit programming language. While this may look like a step backwards compared to implicit programming, we argue that the basic commands to move a robot and grasp/release objects are fundamental knowledge to everybody who is given the task of programming a robot. So we require a basic knowledge of robot movement commands from the target group of this work. In the scope of this work we will investigate how much knowledge in sensor data processing will be necessary.

Chapter 3

Retrieving Information from Sensor Signals

External sensors are indispensable for robots to react flexibly and adaptively. In order to find an approach to sensor integration that is neither fitted to a specific task nor a sensor, we must analyze what kind of information may be provided by the sensor at all. This must be performed on an abstract level to maintain universality.
In this chapter we analyze types of information that may be contained in a sensor signal and describe a systematic approach to recognize changes in the workspace using external sensors, regardless of the type of sensor. We motivate the need for a universal classification of information provided by external sensors in Section 3.1 using an example task and analyze handling tasks on an abstract level. In Section 3.2, we give an overview of existing approaches and delimitations to our approach. In Section 3.3, we show that all sensor information can be classified into two groups: conditions and Cartesian values. In Sections 3.4 and 3.5 we analyze approaches to create functions to extract relevant information from the sensor signal easily. In both sections, we show that these functions possess universal characteristics. These can be used to generalize their practicability for later utilization into the programming concept that will be derived in Chapter 4. In subsections, we describe how these functions may be learned and optimized iteratively during execution of the task. We will evaluate the proposed concept in Section 3.6 and summarize this chapter in Section 3.7. The results of Sections 3.5 and 3.6 are published in [40] and [42].

3.1 Motivation

To motivate the problem of sensor based robot programming, we look at an example task taken from the industrial domain and illustrate the problems that require external sensors.

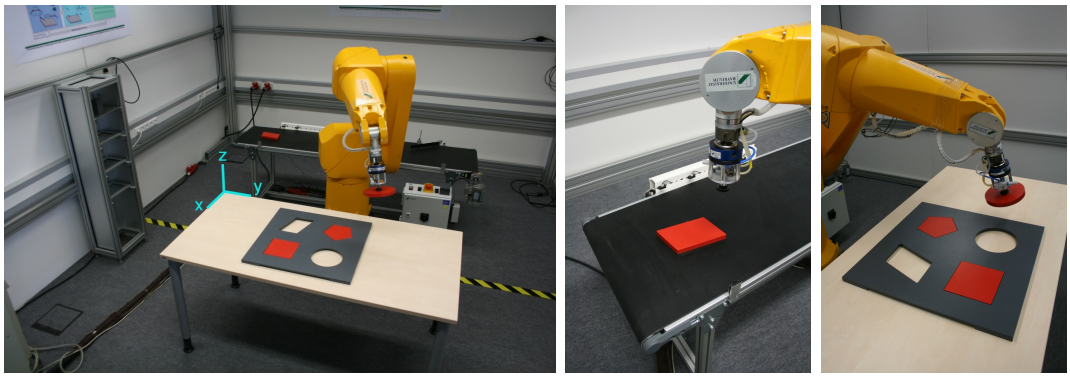


Figure 3.1: Left: Overview of the example task described in Section 3.1.1. Middle: The robot shall grasp the objects from the conveyor belt. Right: The object shall be inserted into the corresponding slot.

3.1.1 Example Task

Consider the following task: A robot shall insert four differently shaped discs into corresponding slots of an object as illustrated in Figure 3.1. The disks are delivered to the robot on a conveyor belt that stops when the disk passes a light barrier so that it can be picked up by the robot. The robot shall insert the disk into the corresponding slot. The world coordinate system is chosen so that the conveyor belt and the insertion area are lying in the x-/y-plane with the z-axis facing up.
Assuming that the discs are delivered to the robot in a predetermined order and that all positions are fixed and precise, a robot program to insert a specific type of disk is relatively simple and will look like this:

Pseudocode 1 (Example task)

```
1 PROGRAM insert_specific_disk()
2 {
3   MOVE p_belt;
4   GRASP;
5   MOVE TRANS(0,0,100):p_insert;
6   MOVE p_insert;
7   RELEASE;
8 }
```

There are two positions in this program: The position `p_belt` describes the position of the disk on the conveyor belt once it has stopped. The `p_insert` describes the position of the corresponding slot in the object. Firstly the robot moves to the conveyor belt and grasps the disk (lines 3 and 4). Then the robot moves 100 mm over the object (line 5) and inserts the disk (lines 6 and 7). The programs to insert the other shapes are similar with the same position `p_belt` and different positions `p_insert` for each shape.

As outlined in Chapter 1.4, the focus of this work is not to determine new ways to intuitively create the program but to enable the robot to act more flexibly. Because of this, here we assume that this program has already been created by the developer. As this program is very simple and neither contains conditional jumps nor repetitions, it can be regarded as a simple sequence of commands, which is relatively easy to program. For more complex programs intuitive ways of creating these must be found as well. This is not the focus of this work. For more information on creating static robot programs see Chapter 2.
The position `p_insert` is different for each disk. A developer could create four different programs for each type of disk. In theory, the complete program could be strung together by these four programs assuming the order of the discs' arrival on the belt is fixed. Unfortunately, there are some impacts caused by external factors in the workspace:

1. The disks are delivered to the robot at a random order. But no shape is delivered twice before all other three shapes have been delivered, so the robot is always capable to fill the whole object before the next object will be processed.

2. The orientation of each disk is not constant. Each disk may be rotated arbitrarily around its z-axis by any degree.

3. Due to deterioration in the feeding mechanism, the pickup position drifts slightly along the negative x-axis of the conveyor belt. So every time a disk is delivered, its position will move slightly compared to the previous disk. This deviation is minimal and will only be measurable after a significant number of executions.

4. The general orientation of the tray where the disks shall be inserted is constant in each execution. So we know the round disk must always be inserted at the top right corner of the tray and so on. But the tray's position on the workbench is not fixed. Therefore the exact location of each slot is unknown.

5. Because the allowance of each disk and the corresponding slot is very low, the insertion procedure in lines 3 and 4 will only succeed if the robot grasps the disk at its exact center and the rotation of the disk is aligned to that of its slot. Otherwise the insertion will fail due to a jamming of the disk.

The problems described in this task comprise some of the most typical applications for sensor based robots and their problems:

- The task of *pick-and-place* with the problem of object localization
- The task of *selection* with the problem of object classification
- The task of *peg-in-hole* with the problem of sensor-based and controlled movements

As outlined in Section 1.1 these problems shall be overcome by installing external sensors in the workspace instead of creating fixtures and feeding / damping devices. For the task presented here, three different kinds of sensors shall be employed:

- A distance sensor (see Figure 1.2, third from left) placed at the side of the conveyor belt to monitor the disk's position on the belt.
- A camera (see Figure 1.2, first from left) supervising the pickup position `p_belt` to classify the disks by their shape.
- A force-/torque-sensor mounted to the wrist of the robot (see Figure 1.2, second from left) to locate the general position of the corresponding slot and then insert the disk.

In summary, we can say that the task presented here is representative for the area of handling in industrial contexts. Creating a feasible solution for this task is not trivial especially for the target group of this work: Developers with only limited expertise in robot programming. So there is a substantial need for the development of a way to incorporate external sensors into a given robot program that can be accomplished by non-experts.

3.1.2 Information Contained in Sensor Signals

To successfully deal with the problems mentioned here, we need external sensors to identify changes in the workspace so we can compute a reaction to it. The first task is to find a function that transforms sensor values into a universal description of the change. While this is a straightforward process for 'easy' sensors, e.g. distance sensors, it proves to be a lot more difficult for complex sensors, like those dealing with images from cameras. Additionally, the sensor values are nearly always corrupted by some kind of noise.
In order to use sensor signals to modify the robot's behaviour during execution, the relevant information must be extracted from the signal. The problem is that this is dependent on three factors:

- The type of sensor used: Imaging sensors such as cameras provide us with a different signal than simple distance sensors in terms of dimension, meaning, and others aspects.
- The type of task: Force/torque sensors can be employed for insertion tasks but are practically useless for object supervision because the sensor has to be in touch with the object.
- The placement of the sensor in the workspace: A camera that is used to locate objects will provide us with different views of the object depending on the viewpoint of the camera. Because of this the detection algorithms must be parametrized with respect to their position in realtion to the supervised object.

The classical approach is to analytically determine a function describing this transformation. But, for complex sensors this task turns difficult very fast and sometimes finding an analytical solution is simply not possible if the underlying physical principles are unknown to the programmer.
We can see that it is up to the developer of the robot program to create these functions

because no general approach can be given. The developer must describe how this information is to be extracted from the sensor signal. But knowledge in sensor data processing is required to achieve this. As a logical consequence we must find ways to enable a developer with no special knowledge in this area to create these functions.
In order to take a general approach on the creation of adaptive robot programs, we must find a universal description for all possible workspace changes in industrial handling tasks. Otherwise each robot program will be designed customly for a specific task. Even if universal algorithms for flexibility and adaptivity can be employed, these must be specifically programmed by the developer in order to use the information contained in the sensor signals.
If there is a universal description for all possible alterations, we can use the following approach: In a first step, the sensor signal is transformed into this description. In the second step, the robot program works only with this description, allowing the developer to employ universal algorithms for flexibility and adaptivity.
Note that we neither deal with the task of selecting a suitable sensor and installing it in the workspace nor with the problem of accessing the sensor signal within the robot program. The first problem depends too much on the task while the second is a problem from the domain of electric engineering, where device interfaces must be specified, etc.
In summary, the purpose of this chapter is to find answers to the following questions:

- What types of workspace alterations can be observed by an external sensor?
- Is there a universal approach to classify all sensor signals according to the information contained in them?
- How can sensor information be used in a robot program regardless of the type of sensor and without limiting the approach to certain tasks?
- What knowledge in sensor data processing is required by the developer to transform a given sensor signal into the universal description?

3.2 Related Work

The task of inferring information from noisy sensor data is covered thoroughly by various books on pattern classification, e.g. [23], [47]. But all of these describe methods how to extract the relevant information from the sensor values, assuming that this information is somehow present in the data. Multiple papers deal with the task of planning sensing strategies for robots, e.g. [79], [102]. Most of these assume a specific task [15], [56] or are aimed at employing multi-sensor strategies [29], [46]. Various papers deal with the task of setting up the sensors in the work cell to allow information retrieval [63]. Kriesten proposes a general platform for sensor data processing, with the drawback of assuming that the sensors are already capable of detecting changes [12]. Papers covering the topic of employing sensors for robot tasks from a general point of view are [50] and [97]. There is one paper

dealing specifically with drift in the servo motors of robot joints [67], but again, this work is geared towards a specific type of sensor.
Two types of sensors are typically used for manipulation tasks: Force-/torque sensors and cameras. When force-/torque sensors are employed, maps are created describing the measured forces with respect to the offset to the goal position. Chhatpar describes possibilities to either analytically compute or create these maps from samples [33]. Based on this, Thomas shows how these maps can be computed using CAD data of the parts involved in the task [117]. In both cases, the maps must be created before the actual execution of the task and they are only valid if the parts involved are not subject to dimensional variations. When the information is acquired by use of cameras, the first step is to perform some kind of pre-processing of the data to extract the relevant information. To determine in which way this information relates to a positional variation is once more task of the programmer and highly dependent on the type of the task. Examples are given in [48], [93] and [125]
In summary, all of the papers mentioned above either propose specific solutions for given types of sensors, tasks or algorithms to extract the relevant information from the sensor data. Neither is there a universal approach for sensor integration nor are any of the solutions presented aimed at non-experts.

3.3 Classification of Information Contained in Sensor Signals

In order to maintain a universal approach to create flexible and adaptive robots, we classify handling tasks and provide two measures to group the sensor information into distinct groups. This classification will be used later to create a well-defined interface for the developer. In order to make this process easier for non-experts, we outline how both types of properties fit into a universal description. This will be used later on in the programming concept to create a general approach to flexible and adaptive robot programming.

3.3.1 Definition of Handling Tasks

In a first step, we take a closer look at handling tasks. At this juncture we advert to the German norm VDI2860 [64], that defines such tasks for industrial contexts. The norm VDI2860 is pseudo-normative, but is sufficiently adhered to by European manufacturers, so it will do for our purposes.
The norm defines the term *handling* as a sub-task of the group *effect material flow*. Other sub-tasks are *shelving* and *transporting*. The difference to shelving and conveying is that handling also incorporates rotatory degrees of freedom. The norm defines the term handling as follows:

Definition 1 (Handling) *Handling is the creation, defined modification or temporary maintenance of a specified spatial layout of geometrically defined objects in a reference*

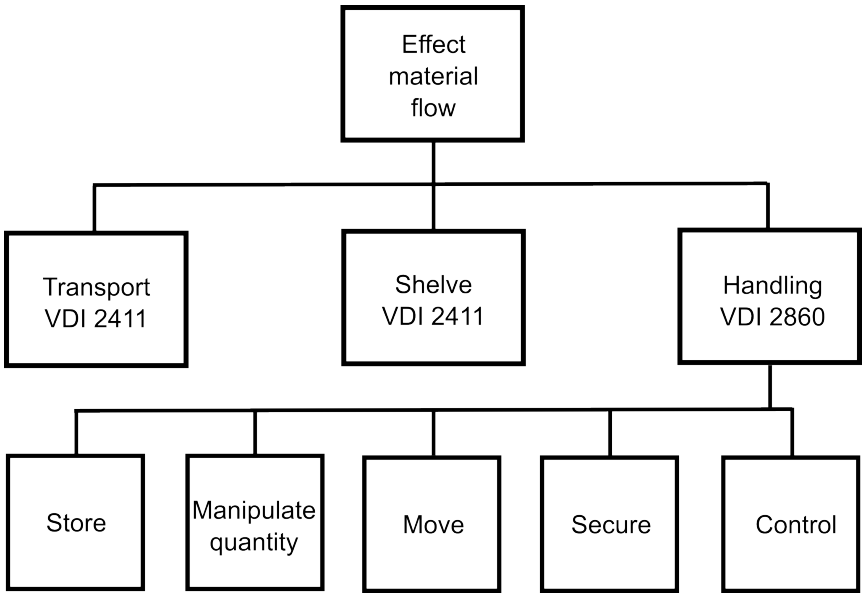


Figure 3.2: Overview of the operations defined in the VDI norm 2860 for handling in industrial contexts. The norm is a specialisation of the general operation *manipulate object flow* which consists of two more specialisations: *Transport* and *shelve* that are both defined in VDI norm 2411.

coordinate system. Additional conditions - such as time, amount and trajectory - may be given.

Handling is further divided into five subgroups. *Elementary functions* are defined for each of these groups. These functions constitute atomic operations that cannot be split into further functions:

1. *Store*: Objects are placed in containers or other storage devices for later retrieval. This group contains no elementary operations.

2. *Manipulate quantity*: Sets of objects can be manipulated by dividing and combining sets. The elementary operations are *dividing* and *combining* sets.

3. *Move*: Objects are manipulated with respect to their spatial arrangement. The elementary operations are *rotating* and *adjusting* objects.

4. *Secure*: Objects are secured to maintain spatial features. The elementary operations are *holding* and *releasing* objects.

5. *Control*: Objects are checked to establish features. There is only one elementary operation, *checking*.

When we take a closer look at these groups, we can see that only three groups are of importance for this work: Moving, securing and controlling.
In *moving* and *securing*, the robot must be able to compensate imprecisions. Here, a geometric displacement must be compensated. This displacement can either occur in an object held by the robot or in the workspace, when the robot (or a held object) interacts with it. In *controlling*, the robot must inspect objects to check if certain properties hold or not. In

this work supervising is used in the sense that the robot employs sensors to check if an object satisfies a given property or not. These properties modify the further execution of the task at hand. At this point it is necessary to distinguish between signals and properties:

- A *signal* is a Boolean value accessible in the robot system that can be evaluated by given commands in the robot programming language. For example a light barrier emits a signal if an object passes. The signal is connected to the robot system and can be accessed anytime during execution to create branches in the robot program. Signals are a standard in modern robot systems and all robots by major manufacturers are equipped with signal handling capabilities. Signals are of importance when the robot program is created, because they are used to lay out the general behaviour of the robot. An example for a signal is the stopping of the conveyor belt by a light barrier in the example task described in Section 3.1.1. When the conveyor belt stops a signal is sent to the robot. The robot will wait for this signal before it continues grasping the object.

- A *property* on the other hand describes a characteristic of a part of the robot's workspace that is measured using an external sensor. It does not influence the general program flow of the robot in the sense that different actions are to be taken if a property holds or not. An example for a property is the classification of objects with a camera in the example task in Section 3.1.1. The robot will only alter the insertion position according to the shape, but will not conduct different actions depending on the disc's shape.

The subgroups *store* and *manipulate quantity* are of no importance for this work. This is because either there are no elementary functions (storing) or they are only of interest for the general program structure (manipulate quantity).
This leads to the following result: Alterations in the robot's workspace modify the robot's behaviour in the sense that movement, securing and controlling functions must be able to be modified by external sensors to ensure correct and safe execution.

3.3.2 Elementary Information Contained in Sensor Signals

At this point we are aware that there are only three general operations in handling tasks that depend on external sensors to ensure flexibility. The question is what kind of information can be contained in a sensor signal. Not all sensors are usable for every operation. In addition, different sensors provide us with different types of data. Unless we are capable to find a universal way to describe the information inherent in any given sensor signal, all robot programs must be tailored to the specific sensor used for the task.
We now analyze what general type of information from external sensors is required for flexible and adaptive robot programs. The question is why we should use external sensors in a robot program? For this work, we assume that we can classify the answers into two categories:

- To measure a geometric property of an object: The sensor is used to determine how an object has moved or altered from one execution to the next. This can be the location of an object (in relation to a specific position), its rotation or a change in its shape (when flexible or organic materials are handled). All of these can be expressed in (sets of) six-dimensional Cartesian coordinates for position, size, and orientation. This information is used for the elementary functions in the groups of moving and securing.

- To form a decision: The sensor is used to determine if an object satisfies a (series of) condition(s). Classifications fall into this group, since they can be expressed as a series of if-then statements. Another issue is to decide if a search motion has reached its goal. All of these can be expressed as Boolean values: A property either holds or not. This information is mainly used for the elementary functions in the group of supervising, but may be used for moving as well. A specific example will be given in Section 7.5.

In the context of industrial handling tasks, these categories are complete. This means that there is no other type of information that can be contained in a sensor signal that does not fall into these categories. The only information we are interested in are geometric properties of objects to modify positions or general properties of objects to select a corresponding action. If some kind of property p can not be described in a Cartesian coordinate system, it must be a property that does not influence any position in the task. This is because the object will not have changed in a geometric sense, so no position (e.g. for insertion) needs to be modified. If p is a non-geometric property of an object o, this can be described using a Boolean equation: Does o satisfy p? Because of this, all information contained in a sensor signal is either a geometric property or describes the condition of an object.
Geometric properties are the main reason to use sensors. We want to modify a position using the sensor signal to compensate a workspace alteration. For instance the disk's location on the conveyor belt in the task described in Section 3.1.1 can be described as a geometric property.
Conditional properties are of interest as well. For instance, choosing the general insertion position based on the disk's shape is realized by using a shape property. Another example of this is to determine if insertion of the disk into the corresponding slot was successful.
Note that this hypothesis only deals with external sensors. Internal sensors monitoring the robot's state are explicitly not included here. All workspace alterations in industrial handling tasks can be classified with this hypothesis.

3.3.3 Point in Time to Extract Sensor Information

Another way to classify sensor information is to group them by the point in time during task execution when information can be accessed. We classify the answers into three categories:

- A sensor is used *preparatoryly*, if the sensor signal can be accessed and evaluated before the robot performs a movement based on that information. Typical examples

are imaging sensors that monitor objects. The image is used to determine the object's position and a robot moves to this position. Furthermore, there is no need for the robot to perform a movement before the sensor signal can be processed.

- A sensor is used *concurrently*, if the robot must perform a movement that alters the sensor signal. The robot waits to evaluate the altered signal before executing the next movement. Typical examples are force-controlled trajectories along surfaces or force controlled insertion. In both cases, the robot moves along a trajectory or a search path in relation to the current sensor signal. The signal must be evaluated constantly to determine the next part of the motion.

- A sensor is used *subsequently* if the information is available after the robot has performed the whole task. This is mainly used for inspection to ensure correctness of the result and to discard faulty parts.

In industrial handling tasks, only preparatory and concurrent sensors are of importance to allow for flexibility and adaptivity. Subsequent sensors cannot influence the robot's behaviour since the information can only be evaluated after the robot has performed the task. Preparatory sensors are mainly used for supervision, while concurrent sensors are used for search motions and insertion procedures. We therefore will only analyze the first two types in more detail.

3.3.4 Classifying Sensors for Handling

In a next step, we analyze if for every combination of these two properties there is an industrial application where such a sensor is required:

- Preparatory sensors are employed to determine geometric properties e.g. when cameras or distance sensors are installed in the workspace to detect an object's position. Because they are not mounted to the robot their signal can be evaluated anytime providing us with a geometric description of the object's location. This description can be used straight away to move to the correct location, provided that the robot does not block the sensors view of the object.

- Concurrent sensors are employed to determine geometric properties e.g. when force/torque sensors are used for complex insertion procedures. In this case, the robot performs a part of the insertion motion, then stops to compute the next position in the insertion trajectory based on the sensor signal.

- Preparatory sensors are employed to determine conditional properties e.g. when cameras are used to classify objects by their shape or some other property. This can be done without the need for a special robot motion.

- Concurrent sensors are employed to determine conditional properties every time a search is executed. The conditional property in this case is the Boolean decision if the

Availability of Sensor Signal	Type of Object Property	
	Geometric	Conditional
Preparatory	Supervision	Classification
Concurrent	Guided search	Search termination

Table 3.1: Use of external sensors in industrial handling tasks.

search has terminated. Unless this condition evaluates to 'true', the search continues. E.g. this is done using a force/torque sensor for intelligent insertion procedures.

We can fill the matrix illustrated in Figure 3.1 with example applications for every combination. As there are example applications for every possible combination, we must provide (abstract) algorithms to deal with each combination.
In summary, we can say that in industrial applications sensors are either used to determine geometric properties to modify existing positions or to form a decision based on conditional properties. It does not matter if the sensor is used preparatoryly or concurrently. Both cases can occur in industrial handling tasks.
In order to program a robot to react flexibly to its environment, the developer must create these functions to extract the information from the sensor data. We call this type of function *sensor transformation*. The input is a sensor signal and the result is a universal description of the workspace change encountered.
The task of creating these functions can only be alleviated to a certain amount as they are highly dependent on the workspace and the type of sensor used. Nowadays, most manufacturers of industrial sensors provide libraries for this, e.g. imaging software by Halcon [4]. But at a minimum, the developer must parametrize and test these functions. In the worst case, the developer must create the whole function from scratch.
The advantage in the use of sensor transformations is that an additional layer of abstraction is introduced. The program can be designed independently from the actual sensor because all workspace changes are described in abstract terms. Now we may replace the sensor with a different type and - as long as the sensor transformation is correct - no alterations have to be made to the program.
In the next two sections, we describe how functions can be created calculating conditional and geometric properties with only basic knowledge by the developer.

3.4 Conditional Properties

Up to now, we have not analyzed how a developer can design algorithms to extract information from a given sensor signal. In this section, we focus on the definiton of *conditional properties*. In the example task from Section 3.1.1 conditional properties are used to determine the shape of the object and to check if the insertion procedure can terminate or must continue.

As outlined in Section 3.3 a conditional property evaluates the sensor data to check if a certain property holds. One of the most common uses for conditional properties is to determine if sensor guided motions shall be continued or must be stopped, e.g. in controlled movements along surfaces or search motions. Note that in case of a controlled movement the next position of the trajectory is altered by the same sensor as well. But this calculation represents a geometric property and not a condition. This leads to the following definition:

Definition 2 (Conditional property) *A conditional property measures some aspect of an object in terms of a Boolean decision. The property evaluates to true, if the object satisfies that measure. Otherwise the property evaluates to false.*

Regardless of the type of application and sensor all conditional properties take some kind of sensor data as an input and return a single boolean value as output. A conditional property is described as a function

$$f_c : S^n \mapsto \{0, 1\} \tag{3.1}$$

where S is the range of values and n is the dimension of the sensor. Typically $S = \mathbb{R}$ but other values are possible as well, e.g. RGB values for image data, where $S = \{0...255\}^3$ for one pixel in the image.
More complex decisions can be constructed by nesting a series of conditional properties. For example, the exact shape of an object can be assigned to an abstract object type o_i with $i = \{0...k\}$ where $k \in \mathbb{N}$ is the number of different shapes. To achieve this, we use k different conditions `f_k`, each of them only checking if the current object is of type k. Then the classification algorithm is a series of nested decisions:

Pseudocode 2 (Creating complex decisions with conditional properties)

```
 1 int classifyObject(S sensorData)
 2 {
 3    if(f_0(sensorData))          // evaluate condition 0
 4       return 0;
 5    else if (f_1(sensorData))    // evaluate condition 1
 6       return 1;
 7    ...
 8    else if (f_k(sensorData))    // evaluate condition k
 9       return k;
10
11   return -1;                   // no object detected
12 }
```

3.4.1 Creating Conditional Property Functions

In many cases there is some kind of software provided by the manufacturer of the sensor to transform the raw sensor signal into a meaningful description. In case of low-level sensors, e.g. distance sensors providing us with only a one-dimensional value, the creation of

a conditional sensor transformation usually is straightforward and not a complex task at all. Here, the developer must only set some threshold values describing for that range of values a property holds or not.
There is a large number of commercial and open source software especially for the domain of imaging sensors. Examples are the software provided by Halcon [4] and the open source library Camelia [3]. With these packages, developers can specify conditional sensor transformations easily and without the need for detailed knowledge in sensor data processing.
If the function is complex or the general type of the function is unknown, neural networks may be used to learn the function automatically. In this case the developer must only provide a set of examples and classify them. This set is used to train the neural network. The advantage of this approach is that no detailed knowledge is required by the developer. The downside is that a large set involving a high amount of different cases is required to train the network correctly. Another drawback is that it is impossible to analyze the function and compare it to theoretical values or other approaches.
Another approach is to record the signals of all sensors during execution of test cases. These records can later be analyzed automatically for bends and leaps that allow to set up conditional sensor transformations accordingly. Automated methods for the analysis and detection are described in [104].

3.4.2 Adaptive Estimation of Conditional Properties

When conditional properties shall be learned iteratively during multiple executions of the robot's task, we start with a basic sketch of the condition and let the robot update and refine this function based on the experiences of each execution. This task becomes complex rapidly and generally is out of the scope for non-experts in learning and artificial intelligence. Because of this it is difficult to integrate adaptive features for online acquisition of conditional properties into a robot system that shall be programmable by non-experts.
In this section we only give a rough outline to which extent we believe an adaptive acquisition may be possible without the need for expert knowledge. In general, the learning of a function describing a conditional property must be performed supervised. Unsupervised or reinforcement learning requires an evaluation of the solution to rate the current state of the function. In case of robot tasks, this evaluation must be achieved by using external sensors. Either the same that were used to evaluate the function or another set of sensors. This would require the developer to construct even more complex high-level functions for evaluation. Because of this, the only feasible option is to let the developer supervise the robot during training. The developer corrects the robot manually if the function produces a wrong result. All sensor values and the corresponding correct result will be stored in a database. This database is used after every execution to generate a new version of the function describing the conditional property. We believe that the best approach to achieve this is to use neural networks or their equivalent. While this approach has the downside that the developer cannot analyze the function 'by hand' because it remains hidden in the network, the advantage is that the developer also is not required to specify any other details than the size and general structure of the neural network. There are several works

on the design and implementation of neural networks for non-experts, e.g. [16], [92], [58], [124].

3.4.3 Summary

In summary, the term conditional property covers a very wide area of sensor information. All conditional properties transform the sensor data into a Boolean value determining if the property holds or not. These kind of properties are more important when laying out the general program structure or when planning capabilities are required from the robot. Because of the heterogeneity in the range of handling tasks we were only able to give outlines to develop functions describing these properties, but we have argued that this kind of property is only of minor importance for the A-Bot project. We have outlined multiple approaches to enable a developer to create such functions and illustrated why the adaptive learning of such functions is only possible in a supervised way.

3.5 Geometric Properties

Unlike conditional properties, geometric properties describe an alteration of the robot workspace in geometric terms. They constitute the main reason for the employment of sensors in flexible robot programs. In this section, we will define the term *geometric change* and show how this change can be modeled using analytical terms. Based on this, we show which premises must be fulfilled in order to successfully recognize an occurring change during a manipulation task. We describe methods how a compensation function can be determined that computes a position deviation for a given sensor signal. The definitions made here are on an abstract level. We will apply them to the domain of handling tasks later in this work. A geometric change is defined as follows:

Definition 3 (Geometric change) *A geometric change is a spontaneous deviation of a position in Cartesian space between the estimated and the actual position of an object in the workspace between two consecutive executions of the same task.*

This means that we approach the position p_o of an object we believe to be correct during each execution t of a task and measure its deviation Δp_o compared to the previous execution:

$$\Delta p_o = p_o(t) - p_o(t-1) \tag{3.2}$$

This definition refers to the positional deviation of the object in Cartesian space. But we will need a sensor to recognize this deviation. This sensor must not be the same that is used to approach p_o, otherwise we are unable to recognize the change. This can be explained by two examples: In the first example, the position is determined using the internal sensors of the robot. If the object has moved, we cannot recognize this change solely with the internal sensors. Instead we have to employ a second, external sensor to measure if a deviation has occurred. In the second example, we use a force/torque sensor to describe a

force-dependent position. In this case, we can employ the internal sensors of the robot to check if this position has moved.
We need to work with a pair of default values describing the original position and its corresponding sensor value. Otherwise we can not determine if a change has occurred at all. At a later point when this kind of property shall be learned during task execution, we will also need to work with a default position. In reality this default position may be virtual, e.g. set in the origin of the sensor. Then all sensor values will be transformed into a Cartesian description how far the object is apart from the sensor's origin.
We see that the position p_o of an object in Cartesian space is mapped to a (vector of) sensor value(s), s, in the measurement space of the sensor, so we have a function

$$f_{change} : \mathbb{R}^6 \mapsto S^n \tag{3.3}$$

where S is the range of values and n is the dimension of the sensor. This function describes the resulting sensor signal if a change in the workspace occurs:

$$f_{change}(p_o) = s \tag{3.4}$$

To successfully adapt to the change, we must be able to infer the position deviation from the sensor values, that is build the inverse function of f_{change}

$$f_{change}^{-1} : S^n \mapsto R^6 \tag{3.5}$$

with

$$\exists f_{change}^{-1} : f_{change}^{-1}(f_{change}(p_o)) = p_o \tag{3.6}$$

Based on this requirement, we can directly postulate that a physical change must modify the sensor signal. Otherwise we would not be able to recognize a change, that is

$$\exists c \; \forall p_o \in \mathbb{R}^n : \; f_{change}(p_o) = c \tag{3.7}$$

We can easily see that no inverse exists for this function. There is always an inverse for all bijective functions. In addition, if f_{change} is continuous, f_{change} is strictly monotonic as well. If necessary the surjection can be guaranteed by a deliberate constraint of the measurement range of the sensor. The function f_{change}^{-1} is the sensor transformation we are looking for. We can insert the sensor signal into this function and will get the current alteration to the default position in Cartesian coordinates as a result.
Another requirement is that the dimension of the sensor must be at least as high as the *degrees of freedom* (DOF) of the change. Otherwise there can be no inverse for f_{change}.
If we are only taking the physical effect into account that maps a position to a set of sensor values, there is no universal solution for f_{change}. Instead the sensor values for a given position are highly dependent on the object's position in the workspace and the type of object that is to be manipulated. It should be noted though, that there are similarities of the sensor transformation to the Jacobi matrix [91].
Another thing that must be kept in mind is the *signal-to-noise ratio* (SNR) of the sensor for the given object. We can only recognize a change if the alteration of the sensor values for a given deviation is significantly higher than the noise generated by the sensor.

3.5.1 Determination of a Geometric Sensor Transformation

To be able to adapt to a geometric change, we must specify a function

$$f_g : S^n \mapsto \mathbb{R}^6 \tag{3.8}$$

with

$$f_g(s) = f_{change}^{-1}(f_{change}(p_o)) \tag{3.9}$$

so that we can compute an estimated relative change for a given sensor value.

Analytical Computation of f_g

The straightforward way to determine f_g is to work out an analytical solution. But sometimes this task proves to be too complex: While the type of function may be known, it can be extremely difficult to determine a set of parameters, that fit the function well enough to the problem at hand. An example is shown in Figure 3.4 on the right. The image shows the data sheet for a distance sensor. A developer faced with the task of creating a function that takes the current sensor value as an input and returns the distance to the supervised object in millimeters must determine which type function approximates the illustrated curve and fit the function parameters accordingly.

Analytical Approximation of f_g

If we cannot calculate the parameters of f_g analytically, but at least have some idea about the type of the function, we create a training set T containing pairs of the physical change and the sensor value. To build T, we systematically create artificial changes and measure the sensor values for each change. Using this training set, we can approximate f_g, so that it will be close enough to f_{change}^{-1} to ensure a valid guess for an existing change. The accuracy of f_g is then determined by the size of T and the accuracy of each sample in T. The minimum size of the training set is determined by the complexity of f_{change} and is equal to the number of parameters in the change function, although the size should be significantly higher to counterbalance noise in the sensor data.
Once we have created a training set, we can use it to approximate f_g, provided that f_{change} fulfills the requirements mentioned above. To achieve this, we use analytical (iterative) methods to fit a given function to the data that minimizes, e.g. the mean error of all pairs in T. Various algorithms for curve fitting exist, the most popular are the Householder algorithm [60],[87] and the Levenberg-Marquardt algorithm [80], [84]. The problem with these and other approaches is that we must have some kind of idea about the general type of f_g, that is what sensor values are influenced by which DOF of the change.

Estimation of f_g for Unknown Function Types

The problem gets even more complex when the type of function itself is unknown. Here, we will need the same training set as for an analytical approximation. Then we can employ

series expansion, neural nets or equivalent methods to obtain a solution for T. In this case, we estimate the type of function that fits the training set best. For example, we can use *multilayer perceptron* (MLP) networks [22],[101] to implicitly learn f_g. While all of these approaches save us the task of analytically determining the general outline of f_g, the price we have to pay for this is that the size of T increases drastically, because we will need a lot more samples to train the MLP adequately.

3.5.2 Adaptive Estimation of Change Functions

One problem with the three approaches presented so far, is that solutions are fixed and prevent the robot from adapting to changes in the environment. For example, the robot must be stopped and re-calibrated if a drift in the workspace or the sensor system occurs. With our our newly introduced model we show how calibration data for f_g can be computed iteratively during the first executions of the task. These methods can be integrated easily into the programming environment, only requiring the developer to specify a minimum of task-dependent parameters. Additionally, we show how the robot adaptively optimizes the task with respect to execution time based on a steadily improving approximation of the function. We focus on sensors emitting one-dimensional signals, such as distance or force/torque sensors. We do not deal with imaging sensors as this class of sensors usually requires an upstream pattern matching algorithm to distinguish the relevant information from the background data.
Approaches for approximation or estimation of f_g require a training set T. So far, we have assumed that we create T offline before the actual execution. Another option is to create T online, so we let the robot match which change will produce which sensor values during the execution of the task. The actual algorithms to infer f_g from T remain unchanged.
The offline approach has the advantage of providing us with very exact pairs for T, resulting in a very well approximated function f_g. This can immediately be put to use and delivers the best possible results without having to re-train the robot at a later point. The disadvantage is that we need to know in which DOF the change will occur beforehand in order to create a set T that covers all possible changes. If we choose to place no external restrictions on the change, we must deal with six possible DOFs, thus enlarging T drastically.
In case that there is no way to obtain T offline, we must make use of learning algorithms to obtain and classify training data during the actual task execution, called *online creation* of T. Every time we measure a change, we add a new pair to T. In this case we do not have to artificially build the training set. But the main problem with this approach remains that we do not know the correct change for a given sensor value.
The central idea of this section is that f_g is unknown and cannot be calculated analytically or approximated beforehand. Instead, the robot will compute an approximation f_{est} of f_g online during the first executions of the task. Instead of two separate phases, i.e. calibration of the sensor and the actual execution of the task, the calibration process is encapsulated in the execution (see Figure 3.3). The calibration might take longer now, nonetheless the program will work correctly from the very first execution onwards. In addition, the devel-

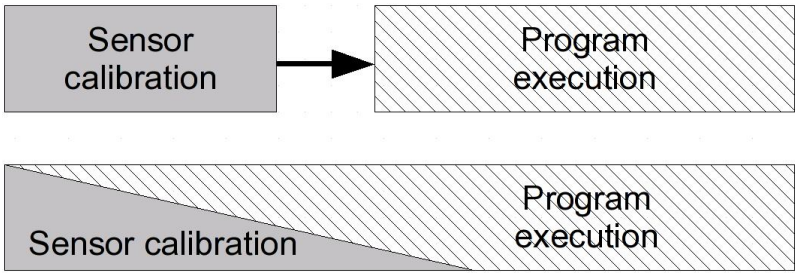


Figure 3.3: In the classical approach to sensor based robot programming, the sensor is calibrated before the actual program is executed (top). In the approach presented here, the calibration process is integrated into the execution cycle (bottom).

oper will spend less time setting up the sensor and the program is capable of adapting to changes both in the workspace and in the sensor data, e.g. due to a warm-up of the sensor, without the need for a manual recalibration. The robot starts with a very rough approximation f_{est} of f_g and refines this approximation gradually with each execution by incorporating newly gained information.
During execution, the robot uses f_{est} to react to changes occuring during the current execution. If the object has moved away from p_o by x to $p_{o'}$, this is detected through the sensor value s:

$$s = f_{change}(p_{o'}) = f_{change}(p_o + x) \tag{3.10}$$

Thus, the robot must modify its movement by calculating:

$$p_{est} = f_{est}(s) = f_{est}(f_{change}(p_o + x)) \tag{3.11}$$

Now, the robot moves to p_{est}. If f_{est} is close enough to f_g then:

$$p_{est} = p_{o'} \tag{3.12}$$

If the change was estimated correctly, this knowledge is incorporated into the change function by adding a new tuple to T. If the estimate was wrong, there is not enough information stored in T to perform a reasonable correction using the current sensor value s. Thus, the correct position must be determined and f_{est} must be modified in such a way that the next estimate will be correct for the current sensor value. This is done by updating the data stored in T. Initially, this will often be the case since early versions of f_{est} are quite inadequate.
At this point we are using a conditional property to validate the geometric change. The conditional property is: Was the change computed correctly or not? This property is easy to program as it only involves subtracting the sensor value after correction s_c from the default sensor value s_d and checking if the difference lies beneath a set threshold c_t.

$$p_c = \begin{cases} 0 & \text{iff } |s_d - s_c| > c_t \\ 1 & \text{iff } |s_d - s_c| < c_t \end{cases} \tag{3.13}$$

When the robot has performed the motion defined by p_{est}, the new position is either correct or it is skewed because f_{est} was not accurate enough. In the latter case, two possibilities

arise. At this moment it is vital to decide whether robot motions will modify the sensor signal or not. This is best illustrated by an example. Consider the following task:
A steel rod is delivered to the robot via a conveyor belt. The belt stops when the rod passes a light barrier (Figure 3.4, left). The robot is to pick up the rod using a vacuum gripper and place it in a box for transport. The rod may be placed in any position as long as it faces upwards. This means we have translational changes along the x-axis and rotational changes around the z-axis in the coordinate system of the conveyor belt. To measure these misalignments, we employ two distance sensors that are placed parallel to the y-axis of the conveyor belt (Figure 3.4, middle). The developer faced with the task to design this robot program now has to plan how the position and orientation of the rod can be recognized and how the robot should react. There are two possible instances:

1. When the robot moves over the belt to pick up the rod, this motion does not alter the sensor signal because the rod itself has not moved. If p_{est} was not accurate enough, the correct position must be searched for. This is usually the case when preparatory sensors are used. The developer can either manually guide the robot to the correct position or instruct the robot to perform an automated search. But it is up to the developer to define a valid search algorithm, because this strongly depends on the task[1]. Once the correct position $p_{o'}$ has been reached, the data tuple $(p_{o'}, s)$ describes a valid data point of f_g, because the sensor value has not changed during the search. This tuple is added to T describing the current knowledge about f_g. With increasing size of T more and more knowledge about f_g is collected and the more precise the next estimations will be.

2. This instance occurs, when the robot has located the rod and grasped it. Now, if the robot rotates the rod, this will alter the sensor signal. In this case a corrective motion can be performed instead of a search. This is usually the case if the sensor is used concurrently. We can employ an automated search: The direction of the search is defined by the Cartesian coordinates that are altered by the sensor. The search terminates when the default sensor value has been reached. Then the robot has corrected the change.

Since this correction alters the sensor signal, we use it to judge the performed correction and compute subsequent corrections accordingly. A correct tuple for T is $(p_{o'}, s)$. Here, we only know s, not $p_{o'}$. But $p_{o'}$ is simultaneously the offset along the x-axis of $(p_{o'}, s)$ from the root, due to the monotonicity of f_{est}. If we perform multiple corrections until we reach the root, we can compute $p_{o'}$ as the sum of all corrections the robot has made. From a mathematical point of view, this is the equivalent to finding the root of an unknown function. At this point we employ the secant method [99], that is defined by the recurrence

[1]Keep the search as simple as possible. As soon as the sensor is calibrated adequately well, the change function's estimate is accurate and always locates the object correctly. So this search is only executed in the very first iterations. Because of this it is not necessary to implement a fast, efficient search strategy, since this represents only a backup strategy in case the change function is still inadequate for a given sensor value.

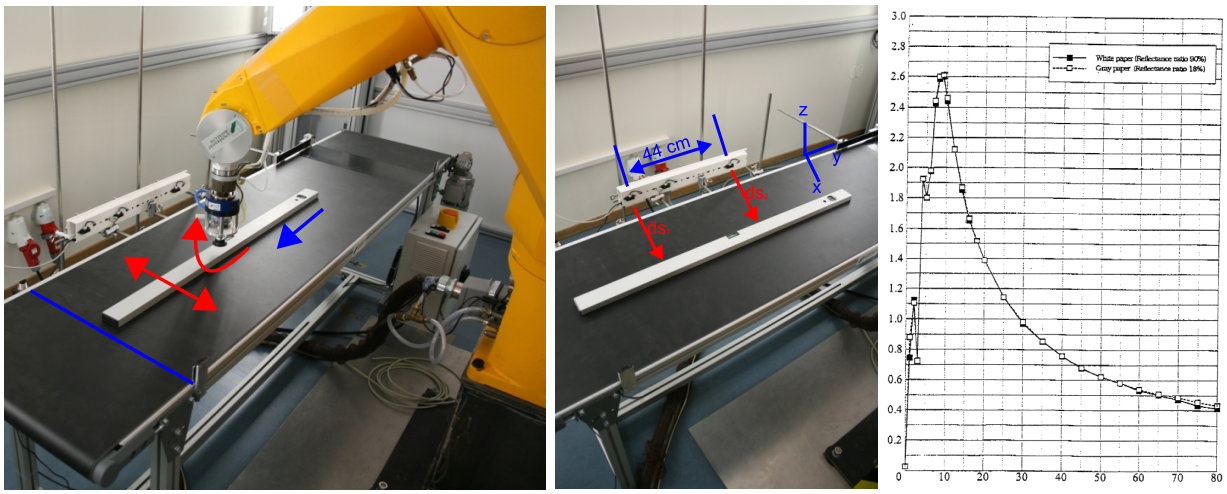


Figure 3.4: Experimental setup. Left: A steel rod is delivered along a conveyor belt (blue arrow) until it reaches a light barrier (blue line). The rod can be in any position on the belt (red). Middle: Reference position of the rod and placement of the distance sensors to recognize the position and rotation of the rod. Right: Scan of the data sheet provided by the manufacturer describing the sensor signal for given distances (x-axis: distance, y-axis: sensor signal). The resolution of the sensor is in the range of [10; 80] cm. To obtain correct values the sensors are placed 10 cm away from the edge of the conveyor belt.

relation

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \cdot f(x_n) \tag{3.14}$$

where f is an unknown function. As can be seen from the recurrence relation, the secant method requires two initial values, x_0 and x_1. The values x_n of the secant method converge to a root of f if the initial values x_0 and x_1 are sufficiently close to the root. The order of convergence is ϕ, where $\phi = (1 + \sqrt{5})/2 \approx 1.62$ is the Golden Ratio. In particular, the convergence is super linear. This result only holds true under some conditions, namely that f is twice continuously differentiable and the root in question is simple and may not be a repeated root. Change functions, as we have defined them, fulfill these conditions.
In our case, f is the real change function f_g and the first value x_0 is simply the change we wish to calculate, $p_{o'}$, while the second value x_1 is the first corrective motion the robot has performed, p_{est} , that is based on the current estimate of the change function f_{est}. Note that f_{est} is used only once for the initial correction, all subsequent corrections are based on the secant method (see Figure 3.5) only using current sensor values. Since the convergence of this method is super linear, we will not need many additional corrections x_n, $n > 1$, should x_1 prove to be poor.
It is important to consider the following: When the next value x_{i+1} is calculated, it must be kept in mind that we have already performed correction x_i before we could measure s_{i+1} to rate x_i. So we must subtract the impact of x_i from x_{i+1}.

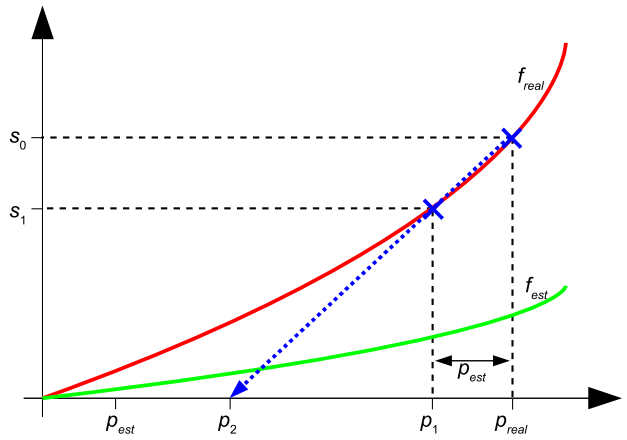


Figure 3.5: Illustration of the first two steps of the correction algorithm: For a given variation p_{real} we perform an estimated correction p_{est} based on the corresponding sensor value s_0, the real change function f_{real} (red line) and our current estimate f_{est} (green line). We move the robot to position p_1 and retrieve a new sensor value s_1. We then use the secant method to grade the last correction and move the robot accordingly to p_2. All subsequent corrections are performed using the secant method only.

Another advantage of this approach is that all corrections x_i and corresponding sensor values $s_i = f_g(x_{i-1})$ are known. We can store these as pairs (x_i, s_i) in a temporary stack. When we have reached p_o, we can use this information to create multiple new data tuples for T. If we have performed i corrections until the robot reaches p_o, the top most pair (x_i, s_i) on the stack already describes a valid data tuple for T. The next pair on the stack (x_{i-1}, s_{i-1}) describes a correction to p_o altered by x_i. So $(x_i + x_{i-1}, s_{i-1})$ is another valid data tuple for the set. Subsequent processing of the stack provides us with a valid data tuple for every correction performed, so we add i new data tuples to T. This leads to an accurate approximation of f_{est} after fewer executions compared to the addition of only one tuple to T in every execution.
The secant method only works for one-dimensional functions. It is possible to combine multiple sensors to obtain an n-dimensional signal. In this case, the Broyden method [27] can be used, that is similar to the secant method. This method is only applicable if a robot motion alters the sensor signal, as is described in Instance 2 above.

Possible Utilization of Other Approaches

The secant method is not the only method to determine the root of a function. Other methods are Newton's method, fixed point iteration, and the bisection method. We will now compare the secant method to these and show why the secant method is the best choice for this task.
Newton's method and fixed point iteration both use the derivative of the function to calculate the next correction. But, as we have explained in Section 3.5.1, it is not always possible to find an analytical solution. Additionally, if this solution were known, it would be more sensible to record a number of examples before setting up the main program and use the examples to determine the function parameters.
The bisection method does not rely on the function's derivative, but has another drawback: To find the root of a function f in an interval $[a, b]$, both $f(a) < 0$ and $f(b) > 0$ must hold, or vice versa. If both values are negative or positive, this method cannot be employed. This is a serious drawback for this case, since we cannot ensure that the first correction we have

Knowledge about f_{change}	Methods for approximation
Thorough / Complete	Analytical computation
General type of function	Analytical approximation
Evaluated training data only	Function estimation
Unknown or no training data	Supervised learning strategies

Table 3.2: Classification of approximation methods to determine the compensation function

performed will result in a new sensor value that has the inverse sign of the first value.
To our knowledge the secant method is the only applicable method that enables a robot to perform a series of corrective motions without any need for backtracking motions by the robot until the root of an unknown change function is reached.

3.5.3 Summary

All of the described approaches are summarized and ordered by their complexity in Table 3.2. If the underlying physical relation between sensor and position deviation is known, the accuracy of the approximation is only dependent on the signal-to-noise ratio of the sensor. In case of online creation of T, estimating f_g with neural nets or similar methods may not be reasonable: In the very first executions T will be too small for adequate training. Later on, we have to keep in mind that every pair in T describes an initial estimate of a change for a given sensor value. The quality of T is determined by the quality of every estimate. When we have to estimate a lot of values, the function will start to fluctuate strongly when many tuples lie closely together. This will deteriorate the quality of T, thus reducing the net's ability to correctly calculate the drift for a given sensor value.
In all cases, f_g must be continuous (or will be, should a neural network be used), otherwise it cannot be approximated by the methods described above.
In summary, the proposed methods are independent of the type of sensor and can be applied to all situations if the sensor is capable of detecting a change.

3.6 Experimental Evaluation

In this section, we will show the validity of our approach by three experiments. In the first experiment, we implement a recognition of rotational variations around an object's z-axis using distance sensors. We determine the geometric change function first analytically and then approximate it using a training set generated offline. Afterwards, we compare both approaches. In the second experiment, we train a robot to react to translational deviations along the x- and y-axis of a disk placed on a table. Firstly, we fit a given function to our training set and afterwards train a neural network to learn f_g and then compare both approaches. In the third experiment, we derive the change functions from the first experiment iteratively during multiple executions of the task.

3.6.1 Measuring the Rotation of a Steel Ruler

We want to find a measurement for the rotation of a steel ruler lying on a table, so we do not have to ensure that the ruler is orientated correctly each time we want to grasp it. For this purpose we use three Sharp GP2D120 distance sensors set up in a straight line facing the ruler (Figure 3.6, left). Each sensor has the resolution of 1 cm in the range of 4 to 30 cm. The sensors are set up 20 cm apart from each other. While it is a straight forward task to determine the ruler's distance from the sensors (something that can be solved analytically quite easily) we are interested in allowing rotational variations of the ruler. In theory, we can measure this variation by subtracting two sensor values from each other and comparing this value to one of the original sensor values. Then the rotation of the ruler is simply

$$f_{rot}(s_i - s_j) = arctan(\frac{s_i - s_j}{d}) \tag{3.15}$$

where s_i and s_j describe distance measurements of two sensors i, j. The parameter d describes the distance that the two sensors are set up apart from each other. Note, that this is the only parameter in the change function. Theoretically, we would determine this parameter and set up an appropriate algorithm to calculate the rotational offset. But here, we will try to determine this parameter experimentally. To achieve this, the robot grasps the ruler, rotates it counterclockwise in steps of one degree and measures the sensor values. This compromises our training set (Figure 3.6, right) of 30 samples for angles of 0° to 30°. To approximate this function we have used a computational approach described in Section 3.5.1 employing the Levenberg-Marquardt algorithm. We have used two general types of functions for the approximation. The first is of the same type as the theoretical offset function f_{rot}, an arctan-function with one free parameter d. The second function is a simple linear function of type $a + b \cdot x$, with two parameters. When we use the sensors installed 40 cm apart, we only use the first 22 of the 30 training pairs. For rotations bigger than 22°, the measured distance of the ruler to the third sensor exceedf the sensor range. The functions are displayed in (Figure 3.7) and the results are summarized in Table 3.3. We have calculated the mean error of each sample to the theoretical value, giving us an impression about the SNR of the sensors. This is relatively low, so we can only recognize changes that are larger than 2°, but the width of the gripper is big enough to deal with this tolerance. To evaluate how well the approximated functions compare to the theoretical function, we have analytically computed the integral error of the difference of the two functions on a range of 0 to 10 cm. We have chosen this range because it is significantly higher than the SNR of the sensors and a difference of 10 cm in the two sensors values would mean a rotation of 26° and 21° for a sensor distance of 20 cm and 40 cm respectively. This gives us an idea of how well we can estimate the function with the given training data. We can see, that if we know that the change function is an arctan function, we can form an estimate with a relatively low error ratio. The error is significantly higher if we do not know the type of change adaptation function and guess it to be linear.
In summary, even with the relatively low resolution of the sensor we are able to enable the robot to flexibly grasp the steel ruler, even if it is displaced by up to 13 cm and rotated

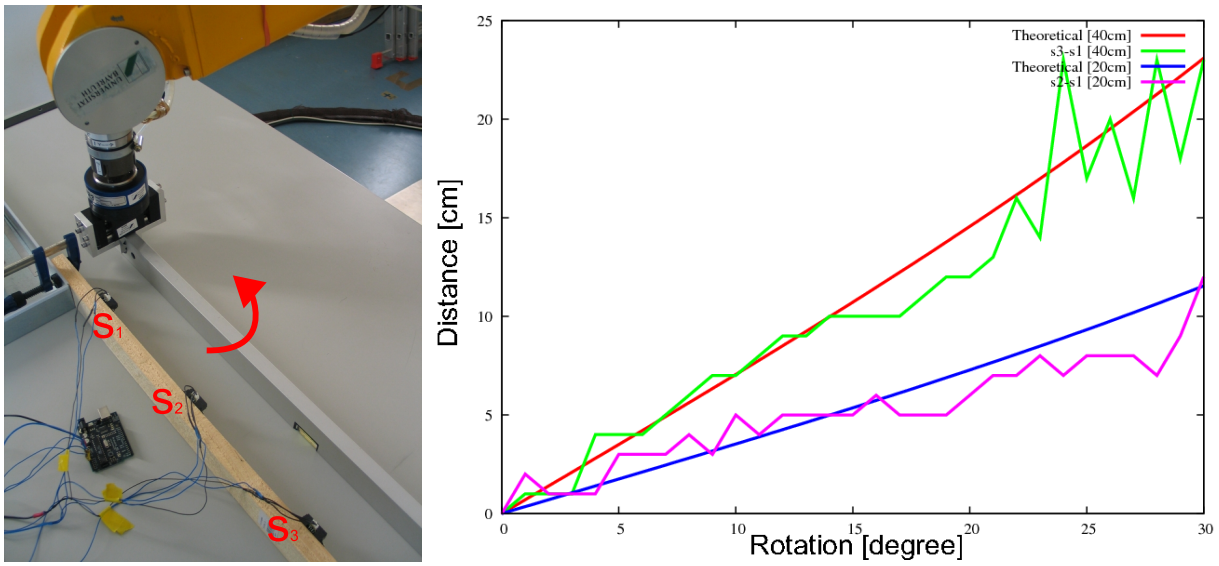


Figure 3.6: Left: Setup of the experiment described in Section 3.6.1. Three distance sensors are used to determine the rotation of the steel rod. Right: Measured distances for given angles and theoretical values. The red and blue lines depict the theoretical change functions for a distance of 40 and 20 cm respectively. The green and purple lines depict the measured differences between the two sensors for a distance of 40 and 20 cm respectively.

by up to 21°. All we have to do is to build a training set by deliberately rotating the ruler and measuring the resulting sensor values.

3.6.2 Measuring the Offset of a Disk on a Table

In this experiment, we use a force/torque sensor to recognize the offset from the center of a round disk along the x- and y-axis of a table. The use of this sensor has the advantage that it is mounted at the robot's tool tip, so no additional sensors have to be placed in the robot's workspace (see Figure 3.8).
The idea is that if the disk moves along the table, we can detect a significant moment along the x- and y-axis because the robot will not grasp the disk in its center. While it is still possible to analytically determine a change function, this is highly dependent on the weight of the disk, its position, etc. Because of this, we try to determine f_{comp} experimentally.
We have chosen to acquire the training set T with an offline approach and have set up a simple algorithm that moves along a grid of a specified size around the center of the disk, picks it up at given intervals and measures the resulting moments.
We know that the relation between the measured moments and the offset is linear and have arranged the general offset compensation function in that way:

$$f_{comp}\begin{pmatrix} m_x \\ m_y \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \cdot m_x + c_1 \cdot m_y \\ a_2 + b_2 \cdot m_x + c_2 \cdot m_y \end{pmatrix} \tag{3.16}$$

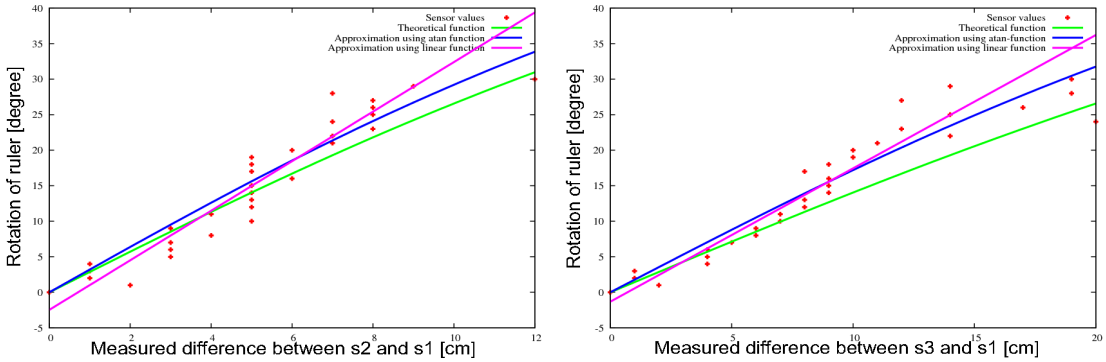


Figure 3.7: Theoretical and approximated functions for a rotational change function. The training data is shown as red dots. The theoretical function is shown in green. The approximated functions are shown in blue (arctan) and purple (linear).

Sensor distance [cm]	Number of samples used	Mean error of raw data to theoretical model [°]	Integral error of arctan approximation to theoretical model [0-10 cm]	Integral error of linear approximation to theoretical model [0-10 cm]
20	30	2.63	1.64	17.83
40	22	2.28	0.92	11.46

Table 3.3: Accuracy of approximated change functions compared to theoretical function for two different sensor distances.

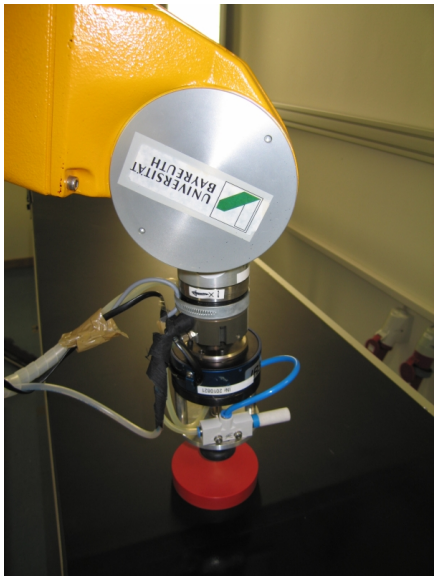


Figure 3.8: Setup of the second experiment. The vacuum gripper grasps the disk at specified offsets from the center and measures the forces and torques for each position. Based on this information the offset compensation function is approximated using a Householder approximation and a MLP neural network.

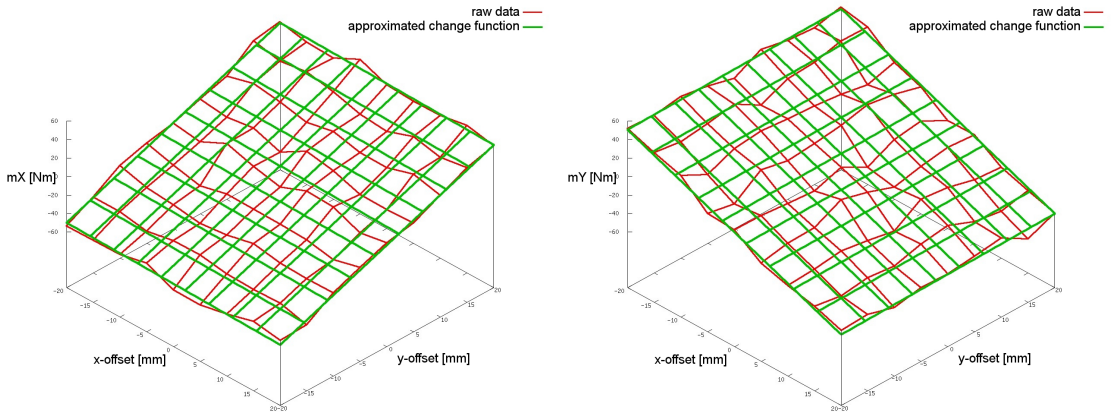


Figure 3.9: Measured moments mX and mY (red) of the disc for given deviations and approximated offset functions (green) using the Householder algorithm for supervised acquisition with a training set of 81 samples.

We assume that a deviation along the x- and y-axis influences the moments along both axes. The sensor values along the grid and the approximated functions for a grid size of 9x9 are shown in Figure 3.9.
We use the same training sets to train a MLP network to learn the same offset compensation function. The MLP consists of three layers with two input and output neurons and three neurons in the hidden layer. Each MLP is trained for a maximum of 100,000 epochs with an desired error of 0.001. The functions learned by the MLP network for a grid size of 15x15 are shown in Figure 3.10.
It can be seen that a deviation along the x-axis mainly influences the moment along the y-axis and vice versa. We calculate the mean error of the approximated function and the training set for various grid sizes, giving us an idea of the SNR of the sensor. To test our offset compensation function, we deliberately move the disk by a random offset p_{real} and record the corresponding sensor values m_{real}. Afterwards, we enter m_{real} into our compensation function and compare the estimated offset p_{guess} to the real offset p_{real}. We repeat this with 100 different offsets for every grid size. The results are summarized in Table 3.4.
We see that due to the SNR of the sensor, that is about 3.5 Nm, we need at least 16 samples in our training set to obtain a reasonable approximation for f_{comp}. Below this value the noise of the sensor prohibits a reasonable approximation. On the other hand, it is unnecessary to create excessively large training sets with hundreds of samples. Above 81 values, there is no significant improvement of the approximation if we increase the number of samples.
In case we do not know the type of function and use a neural network to approximate the function we need more than twice as many samples to approximate the change function with the same accuracy. But for training sets of this size, an equally good approximation is possible.
In both cases the low mean error of the approximations for low grid sizes is caused by

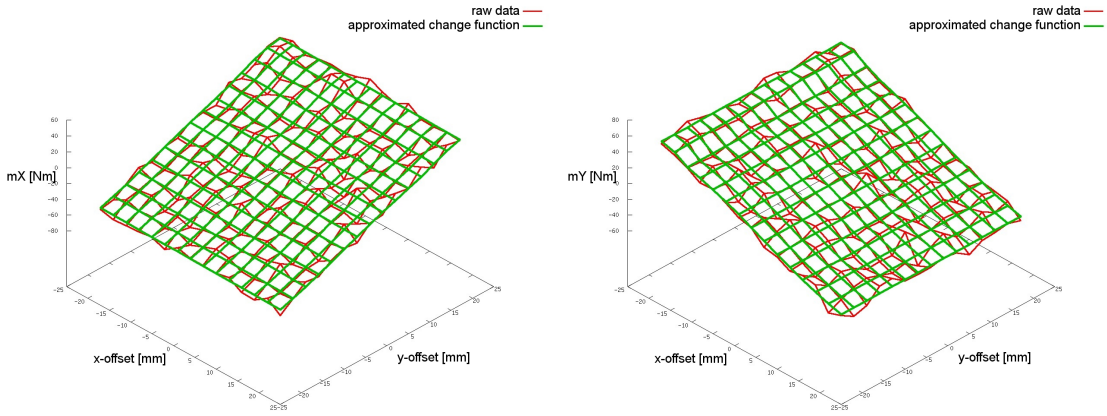


Figure 3.10: Measured moments mX and mY (red) of the disc for given deviations and approximated offset functions (green) using a neural network with a training set of 225 samples.

Number of samples used for approximation	Mean error of raw data to approximated model using the analytical function [mm]	Mean accuracy for 100 random offsets using analytical function [mm]	Mean error of raw data to approximated model using the MLP network [mm]	Mean accuracy for 100 random offsets using the MLP network [mm]
4	3.13	10.99	0.31	12.47
9	3.28	6.33	1.23	8.99
16	3.57	4.03	1.97	7.92
25	3.61	3.57	3.10	4.88
81	4.01	3.62	4.55	4.05
121	4.09	3.60	4.65	4.03
225	4.05	3.62	4.04	3.50

Table 3.4: Comparison of approximation with a given type of function and a MLP neural network. The accuracy of the approximated change functions determined with the Householder algorithm and a neural net for various sizes of supervised training data. The accuracy is determined by testing both functions with samples not used for the approximation.

exactly this fact: There are only few training samples, so it is possible to find a very good approximation with a low cumulative error for all samples, but the mean error for samples not used for the approximation is relatively high. After a certain sample size, the mean error of the approximation remains constant, thus enabling us to determine the SNR of the sensor.
With both approximations, we are able to automatically check if and how far the disk has moved along the table and use the result to adapt to the environment accordingly. The approximations are precise enough to allow for an adaptation without the need to re-grasp the disk.
In summary, we show that it is possible to easily approximate change functions independent of the type of sensor with comparatively little effort. The less knowledge we have entered into the approximations, the bigger the training set must be in order to successfully approximate the function.

3.6.3 Online Computation of Change Functions

We implement the task described in Section 3.5.2. The sensors used are distance sensors GP2D12 made by SHARP with a measurement range of [10; 80] cm. The first sensor supervises the position where the robot is supposed to pick up the rod and measures the translation along the x-axis. The second is located 44 cm away from the first along the y-axis of the belt (Figure 3.4, middle). The difference between the two sensor values describes the rotation around the z-axis.
The data sheet for the sensors shows that the sensor signal is not linear with respect to the physical distance (Figure 3.4, right), so it is not possible to use a simple linear conversion to determine the translation or the rotation of the rod. In theory, the change function describing the rotation can be derived as an Arcus-Tangens function, but the parameters for this function are unknown. Therefore, the robot shall learn both functions adaptively during task execution. A reference position p_{ref} is set up (Figure 3.4, middle), describing the ideal position and orientation the rod should have. This position would be identical to the position of the rod if a feeding mechanism is employed. It is important to measure the sensor value for p_{ref} as well. Later on, all measurements are compared against these values and if it exceeds the SNR of the sensor, a change is recognized.
The robot program for a single task execution is now short and relatively simple:

Pseudocode 3 (Robot program for the experiment described in Section 3.6.3)

```
1 PROGRAM pickupRod() {
2    offset_est = getCorrection(Distance);
3    MOVE offset_est;
4    IF (force_z-axis() < force_contact) THEN
5       searchRod();
6    update(Distance, HERE);
7    graspRod();
8    MOVE p_ref;
```

```
 9    DO {
10       rotation_est = getCorrection(Rotation);
11       MOVE rotation_est;
12    } WHILE (rotation_est != p_ref)
13    MOVE p_dropoff;
14    release_Rod();
15 }
```

In lines 2 and 3 the function `getCorrection` uses the current value from the distance sensor as parameter and moves the robot to the estimated position of the rod. We use a force/torque sensor to check whether the rod was grasped correctly (line 4). This describes a simple conditional property. If this is not the case, we employ a basic search motion probing the conveyor belt in fixed intervals for the rod (line 5). When the rod is located, we manually update T, grasp the rod and move it to the reference position (lines 6 to 8). At this point the rod may still be rotated to an unknown degree. In lines 9 to 12 we correct this rotation by repeatedly calling `getCorrection` until the reference position is reached. The secant method is integrated into this method and remains hidden from the developer. Then we move the rod to `p_dropoff` and release it (lines 13 and 14). Note that the program itself does not contain any sensor data processing apart from the two conditional properties in lines 4 and 12. Additionally, it is neither necessary for the developer to determine the type of the change functions nor any parameters for these functions. To calculate the Cartesian change for an unknown sensor value, we use a simple linear interpolation over all data tuples in T.

We execute the program 100 times. Every time the translation and rotation of the rod is chosen randomly. The initial estimate of both change functions is deliberately badly chosen as a bisecting line (Figure 3.11, top and bottom left). For the change function describing the distance of the rod, we could have also created data tuples using the data sheet of the sensor (Figure 3.4, right). We choose not to do this, for two reasons: Firstly, the data tuples would have to be measured manually by the developer and modified by the distance of the rod's default position, which is a cumbersome task. Secondly, the data sheet is rather small and the resolution is low so it is difficult to determine exact values. Here, it is easier to just use a bad approximation for the very first executions, because this will change after a few executions. Because of this, the robot was unable to grasp the rod correctly during the very first executions and also needed multiple corrections to compensate the rod's rotation. After 10 executions the estimations of the change function look similar to the one in Figure 3.4 and an Arcus-Tangens function respectively (Figure 3.11, top and bottom middle). After 100 executions we obtain a precise interpolation of both change functions (Figure 3.11, top and bottom right), allowing the robot to grasp the rod 20 out of 20 times (100%) without the need for a search motion. The rotation is corrected successfully with just one rotation in 14 out of 20 cases (75%). In the other cases, the robot had to perform more than one rotation to align the rod correctly.

The accuracy of the estimated change functions in locating and rotating the rod during the adaptation process is shown in Figure 3.12. We show whether the robot was able to

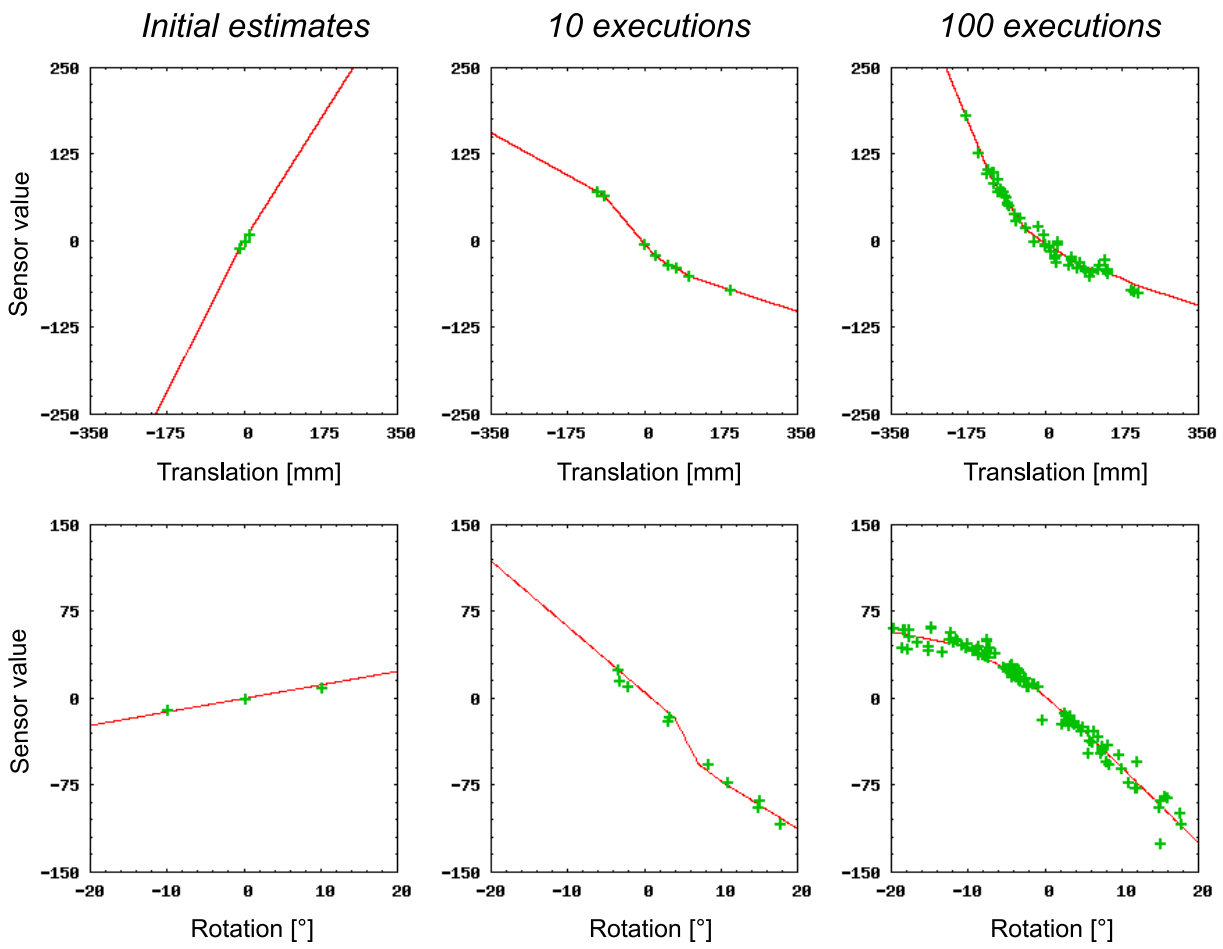


Figure 3.11: Estimates of the change functions to compensate the translation and the rotation of the rod. The top row depicts the change function to measure the translation of the rod, while the bottom row depicts the functions to measure the rotation of the rod. The images on the left depict the initial estimates of both functions, the images in the middle depict the functions after ten executions of the task and the images on the right depict the functions after 100 executions of the task.

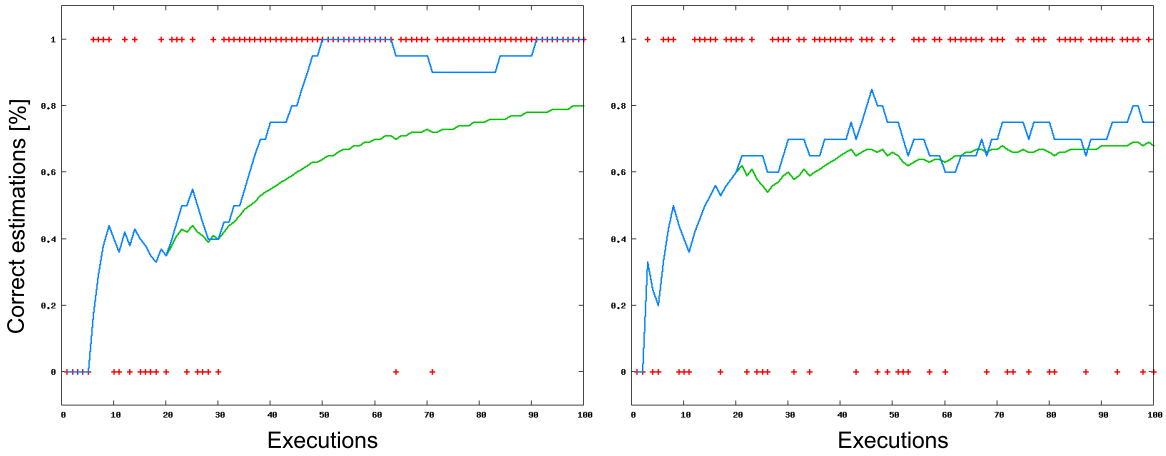


Figure 3.12: Overall (green) and averaged (blue) percentage of correct estimations of the rod's translation (left) and rotation (right) on the conveyor belt using the corresponding change function for 100 executions. A red dot with a value of 0 indicates that the robot could not locate the rod or compensate its rotation directly with the given change-function, but had to perform a search or use multiple rotations instead . A value of 1 indicates that the rod was found without the need for a search motion or rotated correctly at the first try.

grasp the rod and rotate it correctly using the estimates of the change functions (red). A value of 0 means that the robot had to search for the rod or perform multiple rotational corrections, respectively, while a value of 1 means that the estimate was correct. The green lines show the overall accuracy of the robot over all task executions up to that point, while the blue lines show the accuracy over the last 20 executions. We see that the robot was capable of grasping the rod correctly nearly all the time after 50 executions, and had an overall accuracy of 80%. Due to the fact that two sensors are necessary to measure the rotation, the SNR of this combined sensor is relatively high, so the correction could not be performed in one motion every time. In spite of this, the robot was still capable of performing a perfect correction in 75% of all cases.
With this experiment, we have shown that it is possible for the robot to learn geometric change functions from scratch by employing search motions to compensate estimation errors. The use of the secant method to further speed up the learning of such a function can be encapsulated into the programming language and does not need to be programmed explicitly by the developer.

3.7 Conclusions

In this chapter, we laid the theoretical foundations for programming of sensor equipped robots by non-experts. In a first step, we analyzed industrial handling tasks and isolated the operations that may require external sensors in case of workspace changes.

We show that the information gained from sensor signals can be classified into two categories: Conditional properties and geometric change functions. For this work, conditional properties are only of minor importance in relation to geometric properties. A transformation of the sensor specific signal into this abstract description allows us to employ universal algorithms to compensate workspace changes. These algorithms will be developed in the following chapters.
Because of the diversity in the scope of applications it is impossible to define a set of fixed properties that will cover all situations. So this task has to be performed by the developer. We outline various approaches to develop these sensor transformations easily. The focus of this work is on geometric change functions. We present methods to determine this function automatically and order them by the amount of knowledge necessary for the approximation. The presented requirements and methods are independent from the type of sensor.
We present a method to determine this data online during multiple executions of the task. The intention is to develop methods independent from the type of sensor so they can be easily incorporated into a robot program. An additonal advantage is that these methods also adapt to drifts in the sensor signal e.g. caused by warming effects. The sensor signal will change for the same situations. If the function cannot adapt to this effect, the robot will move to the wrong positions. With our method, the data will be modified automatically to reflect this effect, enabling the robot to act correctly.
Finally, we presented three experiments to validate our research. We showed that it is possible to employ the proposed methods to successfully determine change functions for pick-and-place tasks with different sensors. In addition we have shown that these functions can be learned adaptively during task execution with only minor changes of the robot program.

Chapter 4

Simplified Sensor Integration

We have shown in the previous chapter that information contained in sensor signals can be grouped into two categories - conditions and geometric changes - reflecting the desired knowledge we need to encounter workspace changes in robot handling tasks. This classification is independent from the actual task and the type of sensor used. But these functions do not influence the robot's behaviour so far. In this chapter, we develop a framework to integrate sensor transformations easily into a given robot program enabling the robot to react flexibly to changes in the workspace.
The chapter is organized as follows: In Section 4.1 we explain why it is not sufficient to simply add these functions to an existing robot program. In Section 4.2 we examine the different types of change that can occur in the workspace and examine which of these must be encountered with external sensors. In Section 4.3 we measure the potential for adaptation strategies by examining how a given program can be tuned on different levels of granularity in order to identify ways to reduce execution time. Based on these results, we outline our new approach to extend programs with external sensors in Section 4.4 and explain how sensor information is evaluated in this approach. We analyze which class of handling tasks can be solved with our approach in Section 4.5. Section 4.6 summarizes this chapter. The results of Sections 4.2 and 4.3 are published in [39].

4.1 Motivation

Handling tasks differ from manipulation tasks in the sense that there is a stronger emphasis on positions rather than trajectories. The underlying assumption is that it is of minor importance in which way the robot moves from one position to the next as long as it does not collide with its surroundings. While there are applications where the trajectory is also of importance, e.g. complex insertion tasks, it is of more importance to determine the end points of each trajectory precisely enough. This preference of positions yields the advantage that the control cycle does not need to be coupled tightly with sensor data processing. However it is not possible to simply add sensor transformations developed in Chapter 3 to positions. This is due to the following factors:

- The point in time at which the sensor data is recorded and evaluated is critical. E.g. if an imaging sensor is evaluated while no object is in view, the result will be undetermined and the robot's behaviour is unpredictable. In the example task described in Section 3.1.1, the robot may evaluate the distance sensor only when the conveyor belt has stopped. Likewise, the object can only be classified as soon as the robot has grasped it and held it in front of the camera.

- Complex operations like insertion procedures are not trivial. Sometimes the sensor must be evaluated more than once in order to guide the robot through a complex operation. In the example task described in Section 3.1.1, the insertion procedure can not be performed in only one single motion, but subsequent motions must be made based on the current data of the force/torque sensor. Effectively, this will result in a while-loop than terminates when a specific condition is met (namely: The object has been inserted correctly.). This condition is checked by the same sensor but can only be evaluated during insertion.

Of course, it is possible for skilled developers to modify a given robot program to enable the robot to complete the task at hand, but this does not fulfil the desired property of easy to program as mentioned in Section 1.2. In this section, we analyze if the tasks outlined in the VDI norm 2860 can be categorized in such a way that a general outline to integrate sensor data processing into a robot program can be given. We are searching for an abstract approach to deal with changes in the workspace regardless of the actual task and the sensor employed. This approach shall allow the robot to react flexibly and adaptively to the alteration at hand. Criteria to rate the approaches are:

- How strongly are instructions that move the robot and instructions for processing sensor data interleaved in the final program? The idea behind this is that complex programs with interleaving instructions are hard to design and maintain for non-experts.

- The required expertise of the developer. The more knowledge is required by the developer especially about sensor data processing, the smaller the user domain will be.

- The expandability of the concept with different types of sensors. If a specific solution is only feasible with a certain type of sensor, this will reduce the range of tasks that are solvable.

- The applicability of the proposed concept. Are all types of handling tasks solvable with the approach or are there constraints that will render certain tasks impossible?

- The required processing power of the robot system. E.g. if the approach involves a large amount of planning algorithms this may cause the robot to react too slowly for time critical tasks.

We will not analyze other approaches to intuitive robot programming at this point, but refer to Chapter 2 where this has already been done.

4.2 Workspace Changes Across Multiple Executions

In a first step we analyze which changes in the robot's workspace can occur between multiple executions of the same task. Here we are interested in changes that do not require any kind of planning by the robot. That is, the robot is not supposed to try and find new ways to solve the task at hand. Instead we focus on alterations in the workspace that will require external sensors in order to recognize and allow the robot to react to these changes. When the robot is capable of dealing with these alterations we have gained the desired flexibility.
To start with we classify the changes that a robot should be able to deal with and explain that of these can be covered by using sensors. Figure 4.1 shows how these changes can be subdivided into four groups along two characteristics:

1. *The origin of the change*: A change in the workspace can either be caused by the task itself or by external factors. Changes caused by the task describe the desired flexibility the robot should possess. On the other hand a change that is caused by external factors was not foreseen when the task was specified. This change happens unexpectedly. While this kind of change is generally unwanted, it must still be accounted for. The problem with this type of changes is that only rarely an explicit strategy to deal with such changes can be given because of their spontaneous nature. In the example presented in Section 3.1.1, task inherent changes can occur at the general position of the disk on the conveyor belt, the arrival order of the different disks and the exact location of the insertion position. A change caused by external factors is the drift causing the disks to move on the conveyor belt. Another example is if a human moves the table where the disks are to be inserted significantly. In this case the insertion would fail because the insertion position has moved by a significant amount.

2. *The occurrence of the change*: A change can either occur non-recurring or frequently. Non-recurring changes only happen in the very first execution, when the robot does not possess any knowledge about its environment at all or when a significant modification is made to the workspace. Normally, these modifications are only made intentionally, if the program is to be altered. But undesired changes may occur as well, when the environment is altered unintentionally by external factors. Frequent changes happen in up to every execution of the task. These changes reflect the actual need for external sensors, because the robot must be able to detect them in every execution.
   In Section 3.1.1, non-recurring changes are the general position of the disk on the belt and the position of the object the disks shall be inserted to as well as a movement of the table by a human. All these changes occur only once, so as soon as the robot has learned them, they can be regarded as being constant. Frequent changes are the alterations of the pickup and insertion position, the order in which the disks arrive and the drift of the disk on the belt. These changes have to be taken into account in every execution of the task.

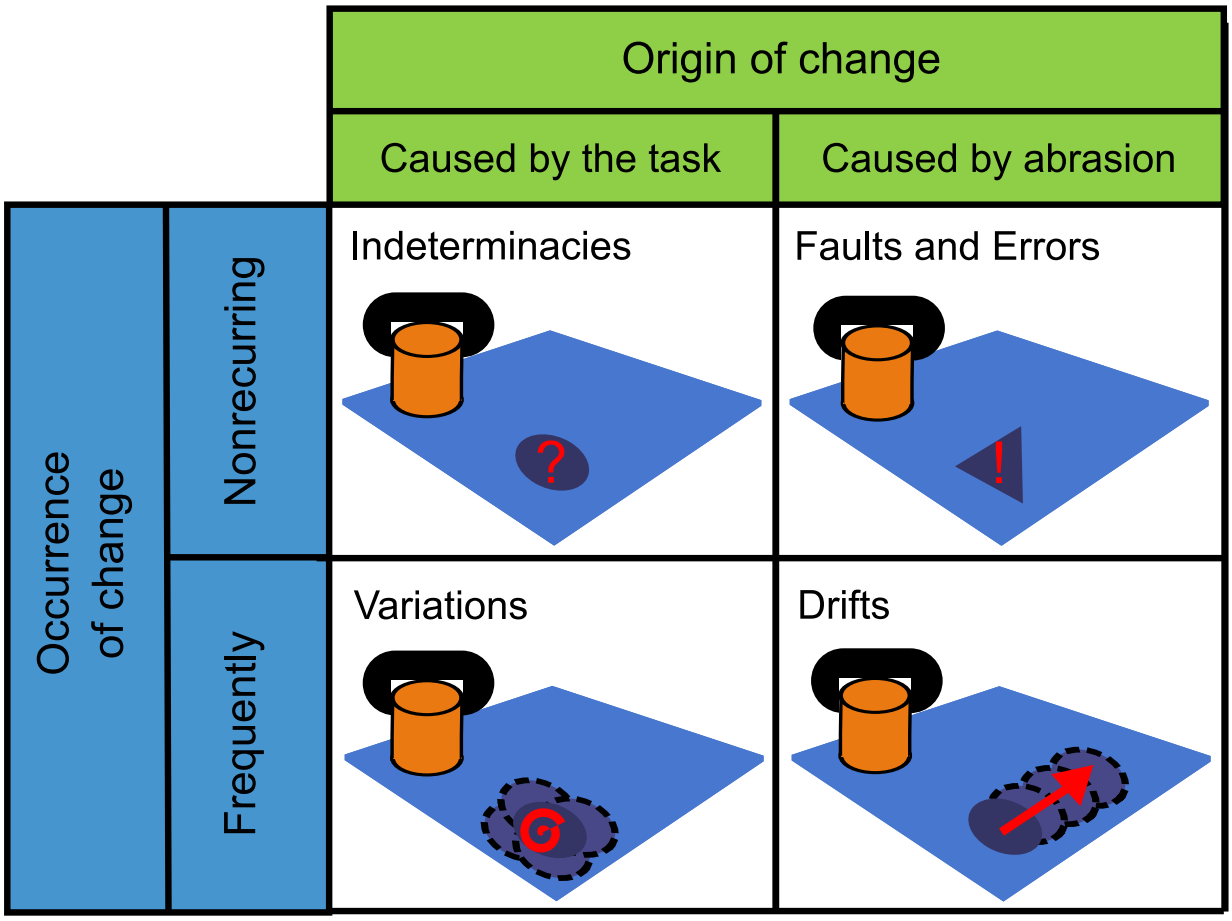


Figure 4.1: Classification of changes that can occur between two executions of the same program

Changes that are brought about by the task itself are *indeterminacies* and *variations*. The difference between these is defined as follows:
An *indeterminacy* is something we are not aware of at that moment, but once we have learned about it, it will remain constant for a prolonged period of time and will not change during subsequent executions. One example is the position of the object on the table in the task described in Section 3.1.1. When developing the program, the programmer usually only knows that there will be a table but not its exact location and orientation in Cartesian coordinates. The robot must be calibrated to learn this indeterminacy (usually by teach-in).
Variations on the other hand occur every time the robot performs the task at hand. They are intentional and occur due to the heterogenic nature of the manipulated objects. This is the main reason why external sensors are used for industrial robot applications. For example, when handling food or other organic materials no object is exactly the same - the objects differ in size, weight, form, etc. In this case, the robot must be able to act flexibly enough to deal with these variations. This can either be accomplished by compliant me-

chanical devices or by incorporating sensors to measure the objects and act accordingly. In the example from the previous section variations are the order and the orientation of the disks on the conveyor belt and the exact location of the corresponding slots in the object. Changes caused by abrasion differ from those caused by the task in the sense that usually they are not intended by the developer; still, they must dealt with. Here, we can subdivide the changes into two groups as well: *Faults and errors* as well as *drifts*.
Faults and errors occur when a sudden change in the workspace takes place. For example a fault would occur if the table in the workspace falls apart due to wear and tear or is moved away by a person. Faults denote an abrupt change in the environment. A common approach is to stop execution and alert a human supervisor if the robot detects that a fault has occurred that cannot be resolved. Other approaches are feasible as well, but no general strategy for error supervision can be given here as this is highly dependent on the task. Every time a fault occurs the robot should inform the supervisor of the task regardless of the fact if the fault can be compensated.
The *drift* is a problem caused by gradual changes within the workspace, i.e. the settings of machines and tools change over time. While this dislocation is minimal from one execution to the next, it can amount to a significant dislocation when it occurs over multiple executions. In our example, the position of the disks is subject to a drift caused by a faulty feeding mechanism. The drift can be modelled as a continuous change within the system. Classical robot programming with fixed positions is unable to deal with this problem. After a certain number of program cycles, either the machines, the tools or the program itself must be re-calibrated by a human operator. Using a continuous adaptation strategy we can tutor the robot to react to this change without the need for recalibration.
Non-recurring changes are only of interest when the static robot program is developed. The developer must identify the indeterminacies and outline an error handling strategy (if any). When all non-recurring changes have been identified, a static robot program can be created telling the robot to move to all indeterminacies in a specified order (with specified trajectories). While the supervision of faults and errors is usually carried out using external sensors, the handling strategy itself must be outlined by the developer.
An error handling strategy is generally laid out as a series of conditional jumps in the form: If this condition holds, we have an error of type X. To resolve this error, perform the following movements.
This condition may be evaluated using an external sensor. Autonomously dealing with faults - that is, recognizing a fault and finding a solution - requires planning strategies, that are out of the scope of this work, so we will not deal with these here. For non-recurring changes the developer himself is more capable than the robot of finding a solution and modifying the program accordingly. Incorporating a complex algorithm for fault detection and compensation would make the robot program exceedingly large and excessive. External sensors may be used to determine indeterminacies (e.g. in sensor based programming, see Section 2.3). But in industrial handling contexts these indeterminacies are not dependent on a sensor in the sense that the robot must evaluate the sensor frequently to determine the indeterminacy. Once the position has been determined, it will not change in subsequent executions (if it does, it would not be a non-recurring but a frequent change,

e.g. a variation)[1]. So, the use of external sensors to determine a non-recurring change is rather a support mechanism than a necessity.
Frequently occurring changes - variations and drifts - can both be handled with a continuous adjustment strategy. They are characterized by the same set of parameters describing the extent to which they are allowed to occur during execution. All changes exceeding this threshold are classified as an indeterminacy or fault, respectively[2]. To deal with this class of changes external sensors is mandatory otherwise exact handling can not be achieved. In sensor based robot programs we should be able to deal with at least variations. Additionally, if we can teach the robot to recognize and compensate drifts, we will be making the overall program more robust.
We see that the general program flow of a flexible robot program is not - or only minimally - influenced by frequent changes, because these are minor deviations from the original indeterminacies in the program flow. So movements based on or modified according to external sensors can be regarded as extensions of a static robot program. They will not modify the general program flow. This is one of the reasons we have decided to take a two-step approach to adaptive robot programming as outlined in Section 1.2.
In summary, changes that occur only occasionally are of interest when the general program structure is laid out. Changes that occur frequently are the main reason to employ external sensors in a robot program.

4.3 Optimizing Robot Programs

In addition to being able to deal with variations and drifts when executing a task, the robot should be able to do this within a certain time frame. That is, it is not sufficient to just be able to adapt to a variation if this will significantly prolong the time required to finish the task. The execution time of the task becomes more important if the task is repeated many times, as explained in Section 1.1.3. It is also of importance to find ways to optimize a given robot program with regard to its execution speed.
How fast is fast enough? This strongly depends on the number of repetitions of the task. If the task is executed only once or a couple of times, the overall execution time is relatively insignificant (as long it is executed correctly). The more often the task is repeated, the more crucial execution time gets. For example, if it takes the robot twice as long to complete the task as a human worker, this is acceptable if the task will be executed only once. In this case it is more crucial that the robot can be programmed quickly, so that the developer can deal with other tasks. But, if the task shall be repeated for a very long time, this execution time is not acceptable, because in this case it will make more sense to let a human worker perform the task. In order to speed up execution time, the program must be

[1]This holds for processing tasks like welding as well. The starting position of the weldseam is usually known. All subsequent motions must be modified using a sensor. While the starting position is an indeterminacy, the subsequent trajectory is a variation.

[2]We will show in Chapter 6 that it is possible to deal with certain types of faults when performing an automatic drift recognition and compensation.

optimized. But this process is time-consuming as well. So before the robot is programmed to achieve a task, the developer must also consider the number of repetitions of the given task.
It is difficult to find a mathematical formulation to determine how fast a robot must execute the task in order to justify its use instead of a human worker. In general, two factors are important: Firstly, how long does it take to program the robot? This is the development time t_d. Secondly, how long does it take the robot to execute the task and how many times will it be repeated? This is the execution time for a single task t_e and the number of repetitions r. The total time to finish the task t_t is then $t_t = t_d + t_e \cdot r$. So if t_t is lower than the time it takes a human worker to finish the task, employing a robot is sensible. But even if t_t is higher, it may still make sense to prefer the robot to a human worker. One argument is that the human is free to perform other tasks. Another is that some tasks are inherently dangerous to humans. For these reasons, no general formula can be given to determine when the use of a robot is profitable.
For any given robot program, we can divide the task of optimizing this program into four levels of abstraction with respect to the execution speed (see Figure 4.2). The higher the level, the more the internal structure of the program will be altered. Note that we only deal with optimization of the program code itself. Neither do we re-arrange the workspace nor do we introduce new mechanical devices, such as better sensors or intelligent robots.
This ordering is strict in the sense that an alteration on a given level implicates that all parameters for optimizations on lower levels have changed and may be optimized as well.
The lowest level (zero) represents the robot program in its original state, i.e. no optimization has taken place yet.
On the first level, we can optimize the acceleration and velocity profiles along the trajectories of the robot. Usually every (transfer-)movement can be divided into three parts: The departure from an object, the transfer itself and finally the approach to an object. Various methods exist to determine the optimal trajectory (and subsequently the corresponding acceleration profile) for two given positions, e.g. [120] and [90].
On the second level we deal with the task of calculating the optimal - either the shortest or the fastest - route to a given goal position. Here we must distinguish between two categories: Open-loop movements and closed-loop movements. In the case of an open-loop movement, the optimal route is the best trajectory between the actual position and the goal position, as there are no external constraints on the trajectory or the goal position. Once again, if the positions are known, we can use existing methods such as in [120] to calculate the optimal trajectory. However, if we need sensor information to accomplish the movement, as is the case in a closed-loop program, we need to optimize the control itself. This is accomplished by minimizing the overshoot in every execution cycle of the controller. This includes vibration-avoiding motions when handling flexible materials. In this case the motion should be executed in such a way that the held object will oscillate as little as possible when the motion has ended. While these motions are controlled by sensors, solutions to optimize these depend only on the parameters of the held object. Closed solutions exist as proposed by [131], [130] and [105].
On the third level of abstraction we optimize the positions themselves. Once again it is use-

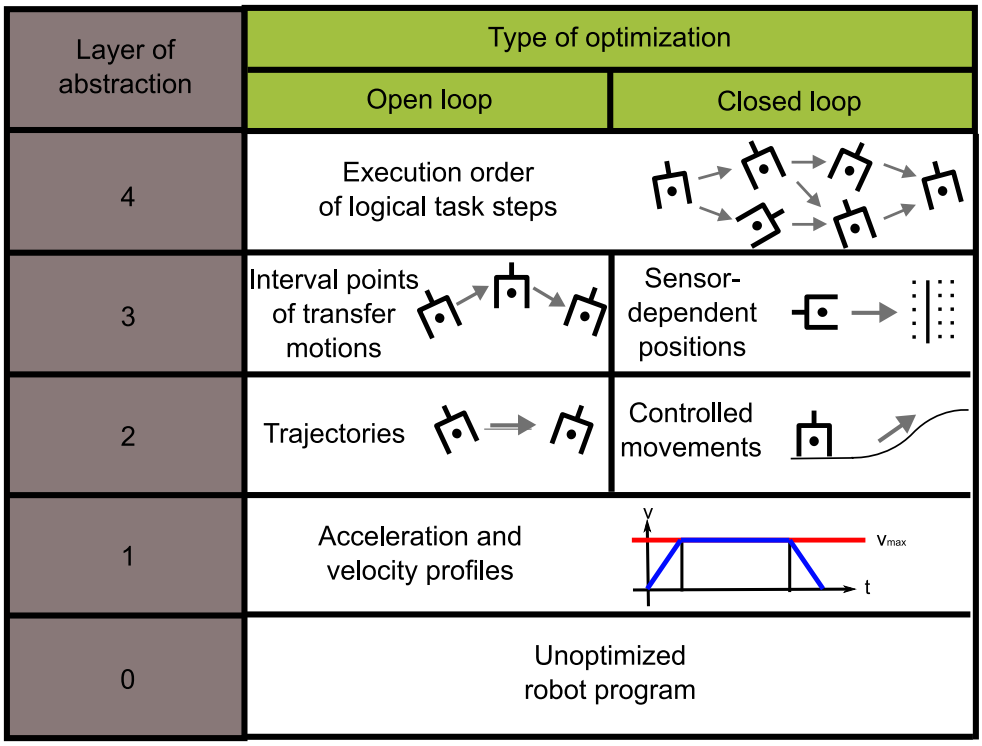


Figure 4.2: Layers of optimization for a given robot program.

ful to distinguish between open-loop and closed-loop programs. If the application-specific positions are fixed and will not change between repeated executions of the program, there is no possibility for optimization. However, it may be useful to determine better interval positions that may result in a shorter overall trajectory to the goal position. In the case of a closed-loop program, the position is dependent on the sensor information available. A typical example is the hole in a peg-in-hole task. Here the robot is guided to the exact location of the hole by a sensor. Another example are movements that are started or stopped by the sensor to compensate for environmental uncertainty. All variations from Section 4.2 can be placed in this field. They can be regarded as a search in n dimensions using an external sensor. If the motion is simply stopped by a sensor, this search is one-dimensional. If the robot shall locate a hole in a plate, the search is two-dimensional along the surface of the plate and so on. There are various possible approaches for optimization: One can either re-calculate the starting position of the search to place it closer to the goal position - thus reducing the time required for the search. Another option is to determine better-suited threshold values for sensor-controlled movements to prevent errors, e.g. a false decision to start or stop a movement. In this case the robot can learn optimal threshold values for the sensor data to start and stop the movement as proposed by [106]. Finally, we can try to optimize the search path. This approach will be pursued in Chapter 5.
On level four of the abstraction diagram we can re-order certain logical parts of the program to reduce waiting time or generate better paths along multiple positions. It may also

be possible for the robot to execute another task while waiting for a machine to finish processing an object. While this level still defines an optimization on the software side of the system, it requires external knowledge about the structure of the task, e.g. which robot commands must be treated as an entity and pre-conditions that must be met before another part of the task can be executed.
It should be noted that the optimization on a high level of abstraction produces the possibility for optimization at lower levels as well. For example, if we compute a new position that is somehow better suited for the task at hand, we can also re-calculate the trajectory leading to that position and subsequently the acceleration profile for that new trajectory.
Up to now the decision to optimize the program is nearly always made by a human supervisor who selects a certain strategy out of the various algorithms mentioned in Section 2.4.2 and applies them to a designated part of the program. Here, we propose that this optimization be performed automatically every time the task has been completed. The developer only marks the areas and levels of abstraction where optimization is permitted while the actual optimization itself is done by the robot system.

4.3.1 Measuring the Potential for Optimization

When deciding to optimize a given robot program the programmer must decide which parts of it should be improved. Usually, he will choose those parts where he believes execution time can be gained. In order to be able to deal with this situation analytically, we have set up some basic experiments for each level described in Section 4.3. The idea is to define a typical situation and a very good (if not the best) solution to it. We will then downgrade this solution by a certain measure and see how much longer the robot requires to reach its goal. We think that in most cases this reasoning works the other way as well. That is, the same amount of time is gained when the solution is improved by the same measure. The goal is to have some kind of chart that tells us for which levels an optimization is auspicious and in which cases the amount of time required to find an optimization may not offset the benefits. All experiments were conducted on a Staeubli RX130 robot with the monitor speed set to 10. The sensor we have used is a wrist-mounted force/torque sensor 90M31A from JR3.
For the lowest level (Level 1) we set up an experiment where the robot moves along a given trajectory using the square wave acceleration profile provided by the manufacturer. The only parameter modifiable for this profile is the maximum allowable Cartesian acceleration. In the experiment, we perform a straight line motion between positions A and B. The goal position B is set to be either 1 cm, 10 cm or 50 cm from the starting position A, respectively. For each of these three motions, we use the maximum acceleration allowable and the maximum acceleration reduced to 90 % and 80 %, respectively. We measure the time it takes the robot to complete the motion and calculate the percentage by which this differs from the fastest possible motion (see Table 4.1).
To measure the impact of a trajectory optimization on Level 2 we set up a point to point (PTP) trajectory between the given positions A and B. (Cartesian motions are omitted since they are always slower than PTP motions.) We then scale the length of this trajectory

Length of motion	Acceleration scaling factor 1.0	0.9	0.8
1 cm	0.271 sec (100 %)	0.287 sec (106 %)	0.288 sec (106 %)
10 cm	0.736 s (100 %)	0.751 s (102 %)	0.767 s (104 %)
50 cm	1.696 s (100 %)	1.728 s (102 %)	1.760 s (104 %)

Table 4.1: Required time for a motion of a given length and a fixed maximum acceleration. The values in brackets denote the change in percent to the reference motion (Factor 1.0)

	Trajectory scaling factor 1.0	1.1	1.2
Time elapsed	2.57 s (100 %)	3.16 s (123 %)	3.40 s (132 %)
Total of joint-rotations	36.1 ° (100 %)	39.2 ° (109 %)	41.8 ° (116 %)

Table 4.2: Required time and total sum of degrees covered by all robot joints during a PTP motion between two points. The values in brackets denote the change in percent to the reference motion (Factor 1.0). The length of the trajectory is scaled by adding a virtual interval position that is covered by the trajectory.

in joint space by adding a virtual interval position C on a continuous path and moving this position so that the resulting overall trajectory from A to B in joint space is stretched by a factor of the original length. We measure the time the robot required to complete the motion and the overall sum of degrees that all joints rotated during the motion execution (see Table 4.2).
The experiment used to measure the impact of an optimization of interval points for complex trajectories (Level 3) is similar to the one described above. However, instead of using a PTP motion, that essentially moves all joints of the robot regardless of external constraints (Level 2), here we typically employ straight-line motions to navigate around obstacles. A typical example is a pick-and-place operation where the robot has to move the grasped object around an obstacle to reach the goal position (see Figure 4.3). The better the intermediate position C is chosen, the shorter the overall trajectory will be. Thus, for this experiment we set up a pick-and-place operation and measure the time it takes the robot to move from starting position A to goal position B using a continuous path motion covering the intermediate position C. We scaled the length of the trajectory by shifting position C and measured the time it takes to complete the motion as well as the total number of joint rotations of the robot (see Table 4.3).

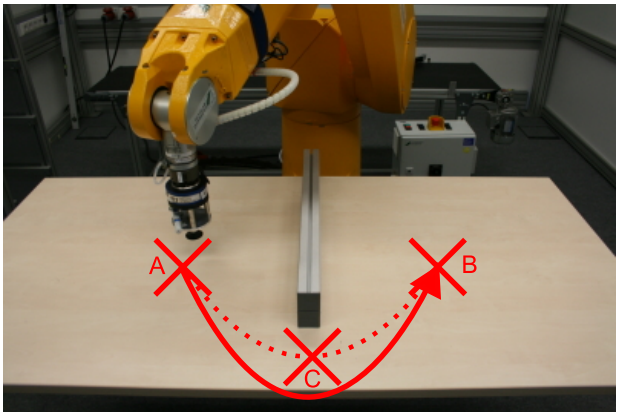


Figure 4.3: Experimental setup for Level 3. The robot moves from A to B, avoiding the obstacle by employing the intermediate position C. For this experiment, we moved C outwards from the dotted trajectory so that the resulting overall trajectory from A to B was elongated by a fixed factor.

	Trajectory scaling factor		
	1.0	1.1	1.2
Time elapsed	2.57 s (100 %)	3.34 s (130 %)	3.47 s (135 %)
Sum of joint-rotations	36.5 ° (100 %)	38.9 ° (107%)	41.08 ° (112 %)

Table 4.3: Required time and total sum of degrees covered by all robot joints during a pick-and-place motion between two points A and B with intermediate position C. The trajectory length is scaled by moving C. The values in brackets denote the change in percent to the reference motion.

	PID parameters scaling factor		
	1.0	1.1	1.2
Time elapsed	16.98 s (100 %)	17.12 s (101 %)	17.20 s (101 %)
Sum of joint-rotations	37.0 ° (100 %)	37.2 ° (100 %)	37.2 ° (100 %)

Table 4.4: Required time and total sum of degrees covered by all robot joints for a PID controlled move along a surface of 50 centimeters. The values in brackets denote the change in percent to the reference motion (Factor 1.0).

	Movement speed scaling factor		
	1.0	1.1	1.2
Time elapsed	62.81 s (100 %)	62.85 s (100 %)	62.83 s (100 %)
Overshoot	-	0.1 mm	0.2 mm

Table 4.5: Required time and overshoot of the tool-tip compared to the original motion for a force-guarded move onto a flat surface. The values in brackets denote the change in percent to the reference motion (Factor 1.0).

To measure the impact of sensor-controlled movements (Level 2, closed-loop), we set up an experiment that moves the robot along the surface of a table at a set distance. For simplicity's sake, we used a proportional-integral-derivative (PID) controller and determined reasonable values for the proportional, integral and derivative parts, so that the movements appear fluid to the human eye. We then scale all values by 10 % and 20 % respectively and measure the time and the total number of joint rotations that occur while performing this sensor-controlled movement (see Table 4.4).
In case of sensor-dependent positions (Level 3, closed loop), there are actually two parameters we can vary: The first is the speed of the movement itself and the second is the threshold value of the sensor, that tells us if we have reached the goal position. As with the other experiments, we employed a force-guarded movement on a flat surface that stops when the force along the z-axis of the tool-tip exceeds 10 N. In the first experiment we scale this threshold and measure the time it takes to execute the motion and the overshoot to the original motion. We repeat each motion ten times and compute the average of both time and overshoot. In this experiment there were no measurable differences between the original motion and the two scaled motions, so we conduct a second experiment in which the threshold value remains unchanged and we scale the motion speed with which we approached the surface to 90 % and 80 % of the original velocity, respectively (see Table 4.5).

On the highest level (Level 4), the execution order of logical task steps, we defined eight

	Route length scaling factor		
	1.0	1.1	1.2
Time elapsed	50.81 s (100 %)	60.20 s (118 %)	59.20 s (117 %)
Sum of joint-rotations	308.8 ° (100 %)	498.3 ° (161 %)	530.2 ° (172 %)

Table 4.6: Required time and total sum of degrees covered by all robot joints for a route along all corners of a cube with a side length of 20 cm. Three different routes were used: The shortest, and two routes that are among the top ten percent and top twenty percent of all possible routes respectively. The values in brackets denote the change in percent to the reference motion.

	Layer of optimization			
	1	2	3	4
Open loop	3 %	22 %	30 %	18 %
Closed loop		1 %	0 %	

Table 4.7: Possible gain on each level of optimization if the corresponding parameters are optimized by 10 %.

positions forming a cube with a uniform side length of 20 cm. The task is to move to all corners of the cube starting at one corner and then back to the start. This experiment is a practical implementation of the traveling salesman problem [37]. For eight points there are 5040 different routes presenting a valid solution. We determined the shortest route and two routes that are among the top 10 % and 20 % of all routes with respect to their length. As in all other experiments, we measured the time and the total sum of joint rotations occurring along the route (see Table 4.6).
We summarize the possible gain yielded by all experiments in Table 4.7. Only the time potential for an optimization of 10 % is displayed. For the potential at Level 1, we used the average of three results.

4.3.2 Conclusion

We can see that an optimization by 10 % on a given level seems to be very profitable for the open-loop parts of the program. However, we have to consider two things: Firstly, the results for Levels 1 and 2 are of theoretical value at best. An optimization of a robot's acceleration profile would require the developer to modify program code at the very heart of the robot system (if this is possible at all) as this constitutes one of the core functionalities of the robot. Acceleration profiles are usually set by the manufacturer because of this reason. Sometimes it is possible to set specific values for the acceleration of each joint of the robot. But this is done in terms of percentage of the maximal value physically possible.

Values cannot be set higher than this. So the best optimization possible is to set all joint acceleration values to the maximum value. No improvement can be made after this. On Level 2 the best motion from one position to another is a PTP motion, if there are no external constraints such as obstacles etc. This motion can be regarded as a straight line motion in the joint space of the robot and is already the shortest joint motion. While the results for the closed-loop parts of the program seem to be disappointing, it must be kept in mind that a good choice of parameters for a given sensor is tricky at best. Unless the programmer is highly experienced in this field or a very thorough analysis of the parameters has been made, the choice of these parameters is usually trial-and-error. While the gain is minimal if an already well-chosen parameter is moved closer to the optimal value, the gain will be much higher if the same parameter was poorly chosen to begin with. We believe that the potential for optimization on these levels is much higher than these first experiments predict. The potential for optimization in Levels 3 (open loop) and 4 is significant and it might be worth the effort to attempt an optimization for these levels.
In summary, we can say that while optimizations of a robot program are possible on many levels, some are either of only theoretic value (acceleration profiles), can be optimized offline by closed solutions (all open-loop aspects) or require extensive further knowledge about the task, e.g. a model of the workcell (trajectories and logical ordering of parts of the task). While there is only little to be gained when optimizing closed-loop motions by 10%, the hard part is to find reasonable parameters at all. Here it is of higher interest to find suitable parameters with only little required input by the developer. The optimization strategies should remain hidden from the developer as much as possible. In Section 5.3.2 we will show how searches (that are parts of variations) can be optimized significantly without any input from the developer.
In this work we will deal with optimizations on Levels 3, closed loop, and 4. As outlined above, Level 1 optimization is made by setting all joint acceleration values to the maximum value allowed by the manufacturer. We will employ PTP motions with no interval positions so no optimization can be performed on Levels 2 and 3, open loop. Controlled movements (Level 2, closed loop) are of importance for processing tasks but only very rarely for handling tasks. In addition, Dauster [36] has developed a method to automatically optimize such movements over multiple executions. We will use that work if controlled movements are required for task execution. On Level 3, closed loop, we are interested in finding sensor-dependent positions fast and accurately. On Level 4, we are interested in optimizing search paths, as the whole path can be regarded as a series of movements that are interchangeable.

4.4 Intuitive Sensor Integration

The central question of this chapter is how external sensors can be integrated intuitively into a given program. Here, we describe how a given static robot program can be extended by external sensors using the results of the analyses from Sections 4.2 and 4.3.

4.4.1 Extending Static Programs by External Sensors

Currently, sensor data processing algorithms are integrated directly into the program. Data acquisition and processing instructions are interleaved with robot instructions.
The advantage of this approach is that the robot is highly flexible and sensor data processing can be optimized along the current robot motion. But the developer must have a very good understanding how sensor data processing and robot motions interact with each other. It takes skilled developers to create such programs. Persons with only limited knowledge in robot programming will have difficulties integrating external sensors in this way.
In modern robot programs we can distinguish between positional information and the program flow. The program flow can be seen as a detailed manual describing how the task is to be completed. The positional information is separate from this flow in the sense that this information describes the place where the task is to be executed. For example, this approach is taken in the *Kuka Robot Language* (KRL) [6]. Because external sensors describe alterations of the workspace and no planning capabilities are required (that would alter the program flow), we take a different approach in this work: Instead of adding more lines of code to the program, we extend the positional information to incorporate sensors. Since all workspace changes are related to objects and their positions, it makes sense to encapsulate sensor data retrieval and processing into the positions. Alterations of the program flow - that is, executing another branch of the program based on a decision made by an external sensor - are specifically not addressed here. If these types of decisions must be made, the developer must modify the program flow. We will explore the use of conditions for a subset of decisions altering the program flow in Section 4.4.4. Here we want to provide an intuitive way to integrate this knowledge without the need for special programming skills. This new concept is illustrated in Figure 4.4.
This approach has the advantage that robot instructions and sensor data processing remain separated, which increases maintainability and allows for a more intuitive concept of programming. Another advantage is that the development is now made by thinking in workspace alterations only. Humans have a hard time interpreting an abstract sensor signal or thinking in robot joint values. This extension of the positional information is more intuitive than abstract sensor data processing algorithms in the main program and is adequate to deal with the imperfections in the workspace encountered here.
We have explained how sensor signals are translated into Cartesian descriptions of changes or are used to check if certain conditions hold in Chapter 3.3. Here, we assume that the alteration of the workspace can be observed by a sensor and that there is some kind of sensor transformation function giving us a description of the alteration in terms of either Cartesian coordinates or object properties, such as shape, colour, etc.
The interaction of all involved components is illustrated in Figure 4.5. The robot system uses a database (*Position Manager*) that stores all positions of the program. A robot position (*Extended Position*) is not only composed of values describing the position and orientation of the robot in a taskframe (*Basic Position*) but also of a number of extensions. An *extension* is a description how a sensor signal modifies the position. Each extension

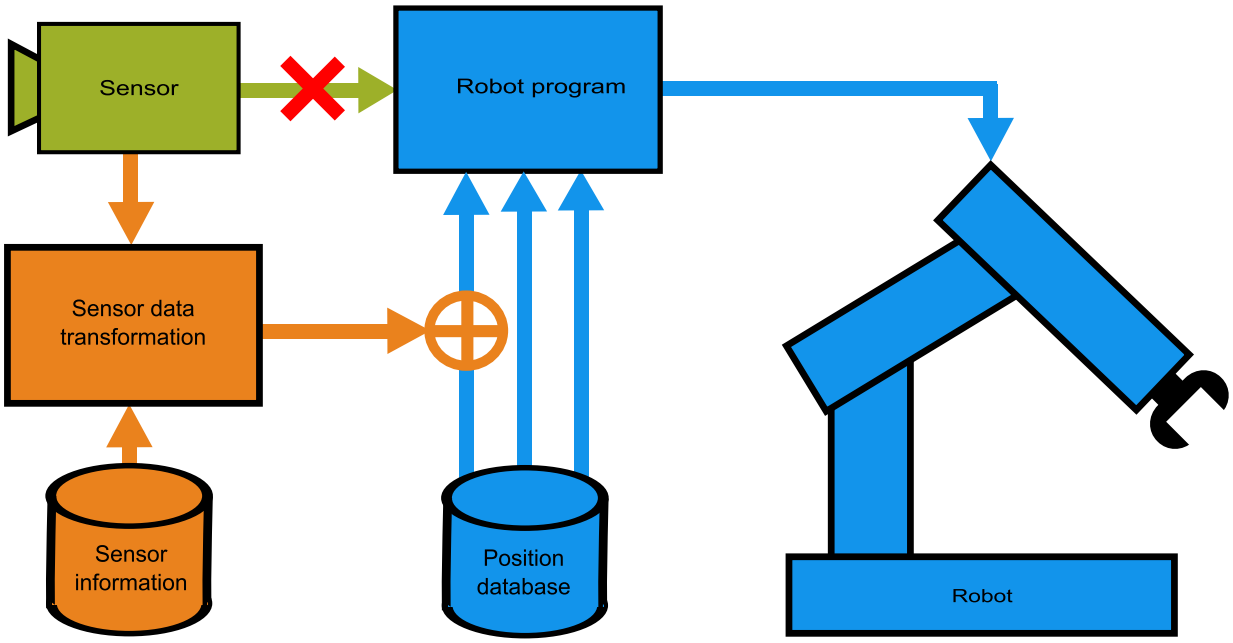


Figure 4.4: Proposed concept of sensor data evaluation.

evaluates a *sensor* and a number of Cartesian *change functions* and Boolean *conditions*. An open question at this point is at which point in time during execution the sensor needs to be evaluated. We will examine various approaches to this in Section 4.4.5.

4.4.2 Applying Sensor Information to a Position

When a robot requests a position from the database, the database checks if that position exists first. In the next step the basic coordinates of the required position are retrieved. Then the database checks if any extensions are attached to the position. These are processed in the order they were set up. The current sensor value for each extension is retrieved and used to compute the modification by the sensor. These modifications are added to the

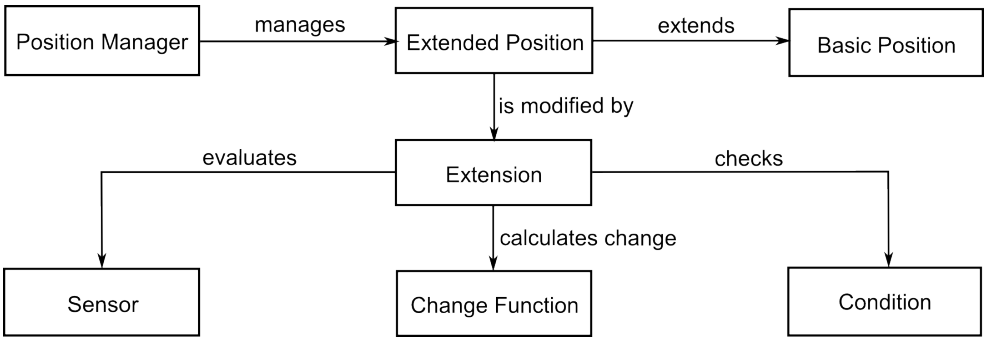


Figure 4.5: Interaction of classes in the proposed framework.

default position. The final result then is returned.
The whole process of requesting a position by the robot and answering the request is shown in Figure 4.6. The sequence diagram shows the process of selecting the position p from the database and applying the modifications according to the current sensor data. The position manager looks up p and determines the default coordinates. This involves accessing the sensor and transforming the sensor data into a position modifier. This is done in a function called `calculateModifier`. Depending on the type of extension, this function performs differently. We will explain this function in detail in the following sections. When all extensions have been processed, p is passed to the `move` command.
This is a universal algorithm to evaluate sensor dependent positions, which is by no means task-dependent. This algorithm can be encapsulated into the programming environment and will remain hidden from the developer. The advantage is that the developer does not need to know how different extensions are processed in the database but only needs to parametrize them.

4.4.3 Modifying Positions Using Geometric Change Functions

This leads directly to the first type of extension that may be attached to a position. Within a *change extension*, we specify a sensor S and a corresponding geometric sensor transformation f. This is attached to a position p. When p is to be approached with a `move` command by the robot, S is evaluated and the signal is transformed using f. The result is a Cartesian description of the current change d. This is applied to p by addition. This algorithm is encapsulated into the function `calculateModifier` from Figure 4.6. A sequence diagram of the function is shown in Figure 4.7. Depending on the type of extension the sensor data is processed differently.
It should be noted, that we do not deal with trajectories. As illustrated in Section 4.3 the task of finding a suitable trajectory between two positions requires a detailed model of the workspace. It is beyond the scope of the actual program to deliver such a model, we assume that - unless special intermediate positions are provided by the developer - the robot will always take the shortest path between two consecutive positions. This is usually a straight line in the joint space of the robot.
With this extension the robot already is capable of localizing objects using the sensor signal. That is, simple changes can be dealt with. Up to now, searches and advanced drift handling techniques, such as drift compensation, require further modifications of this framework. We will explain how these can be handled in Chapters 5 and 6.

4.4.4 On the Use of Conditions in Flexible Robot Programming

Before we explain at which point in time sensor data is evaluated and applied to a position, we examine the use of conditions in flexible robot programs. Because a condition does not encode any kind of geometric information per se, but only a binary statement, we cannot use such properties to modify the robot program directly. So when are conditions used in robot programming? There are three scenarios where a condition may be used:

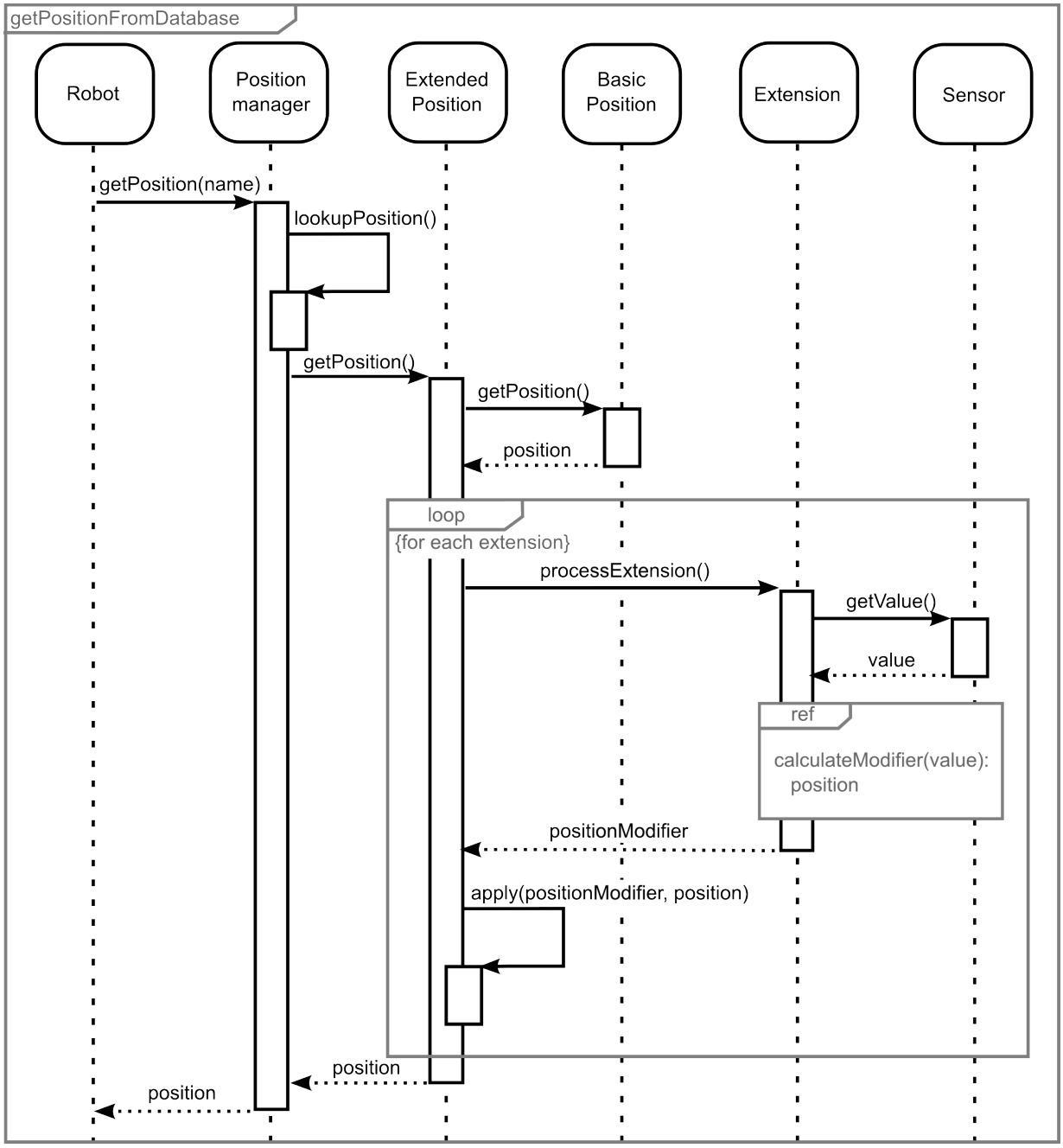


Figure 4.6: Sequence diagram to request a position from the position database.

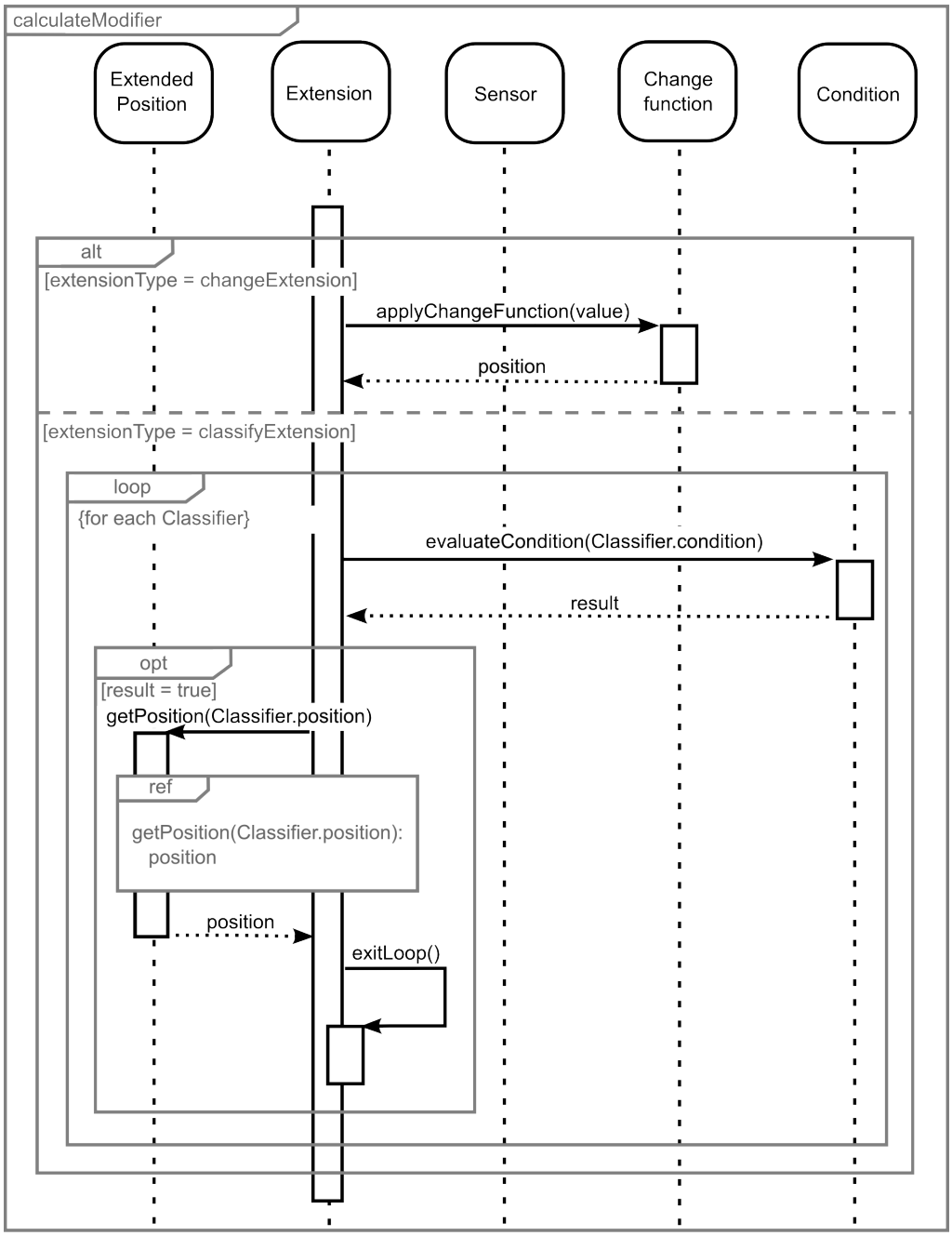


Figure 4.7: Sequence diagram to apply sensor information to a position. Note, that a call may be made to retrieve another position. This call will recursively process another extended position as described in Figure 4.6.

1. To modify the general program flow.
   In this case a condition will be used to select a certain branch of the program and execute different sets of motions depending on the result of the evaluation of the condition. In modern robot programming languages, this concept is called a *signal*. Most robot systems are equipped with wiring capabilities to connect external devices sending electrical inputs to the system. Examples are triggers for light barriers, conveyor belts, etc. So these signals can be seen as simple conditions, in the sense that they also form a boolean decision, but no complex sensor data processing is made to come up with the result. This includes synchronizing the robot with other machinery, e.g. conveyor belts.

2. To select a position.
   In this case a condition is used to determine a property of some object in the workspace and move the robot to a position that depends on that property. Unlike a geometric property, the actual position cannot be inferred from the sensor signal, but is encoded by the developer. So the relation between the position and the corresponding condition must be set manually by the developer. An example would be the selection of the insertion position based on the object's shape in our example task.

3. To end a sensor controlled operation.
   In this case a condition will instruct the robot to either continue or to stop the operation and execute subsequent commands in the program. Once again, the condition will not provide us with any information where the robot shall move, but only with the information that some kind of sequence must either be continued or repeated. An example would be the insertion of the disk in its corresponding slot. Here, a condition would instruct the robot to continue the insertion until the object's position matches the position of the slot.

For our work, we are only interested in the second and third scenario. We assume that the total program flow with all branches has already been laid out, so there will be no additional sets of motions that need to be integrated into the program as outlined in the first scenario. The second scenario constitutes a *classification* based on a condition, while the third scenario constitutes a *search* with a terminating condition.
Searches form a subclass of variations where the sensor is used concurrently. We will explain how these problems are solved in Chapter 5. In this section, we will only outline how classifications can be integrated into our programming concept, as for these the sensor will be used preparatoryly:
The developer must specify a mapping C. The purpose of C is to relate a position p_i to a condition c_i. C is then attached as an extension to a position p_c. The coordinates of p_c are not of importance and may be set to zero. In the program a `move` command to p_c will access C. All conditions are evaluated and the position p_i whose condition evaluates to *true* will be approached. The order in which the positions are added to C is of importance. The first condition that evaluates to *true* will define the final position.

This algorithm is embedded into a new type of extension called *classifier extension*. A sequence diagram of the algorithm processing this type of extension is shown in Figure 4.7. With this extension and the one outlined in the previous section, we can integrate sensors that are used preparatoryly. The position is modified accordingly before the robot will move to that position because the sensor information does not depend on a robot's movement, as opposed to a search motion. If sensors are used concurrently, other more complex approaches have to be taken. Extensions for this and their corresponding algorithms for this are described in Chapter 5.

4.4.5 Point in Time to Update the Position Database

Every time the robot moves in the work cell this will influence the sensor values. Some sensors may only provide useful information at specific points in time. For example a force/torque sensor mounted to the robot's wrist can only be used if the robot is in contact with an object. In other scenarios the robot may move into the field of view of a camera and occlude the object that is to be supervised by that camera. So, it may be crucial to access the sensor at a precise point in time when executing the task in order to obtain meaningful values.
There are four options at what point in time the sensor signal can be evaluated:

1. Permanently at specific intervals
2. When there is a significant change in the sensor signal
3. When the position is accessed in the database
4. When the developer explicitly commands it

In the first, third and last case the sensor is accessed explicitly by a *pull* from the robot system. In the second case, the sensor *pushes* its information to the robot system.
The first option is unacceptable since it will impose a significant computational load onto the robot system. The retrieval interval must be set relatively low in order to detect changes as fast as possible. But for a significant number of sensors the system will be kept busy by just evaluating all sensors and applying the results to all positions in question.
The second option avoids this problem by only sending new sensor information if the signal has changed significantly. But this will also happen if the change is unintentional, e.g. if the robot moves into the field of view of the camera. In this case a decision must be made, if the change is of interest to the robot system, that is if it is a *valid* change. This decision is made by so-called *validity functions* that evaluate the current state of the system and return a Boolean value. Validity functions f_v for a given sensor S, a robot R and a position p are defined as follows:

$$f_v(S,R) = \begin{cases} 0 & \text{iff } R \text{ is in such a state that } S \text{ provides a faulty signal for } p \\ 1 & \text{iff } R \text{ is in such a state that } S \text{ provides a correct signal for } p \end{cases} \tag{4.1}$$

Every time the sensor detects a significant change, the corresponding validity function is evaluated and only if it evaluates to *true* the position is updated in the database. The downside is, that the developer is not only faced with the task of describing how sensor data modifies the position but must also create a corresponding validity function. Another disadvantage is that it is not possible to define such a validity function at all unless a detailed model of the workspace and the robot is available. A good example of this problem is a distance sensor that will only tell us the distance to the next object in its view. We have no way of identifying if the object in question is really the object we want to supervise or the robot that has moved between the sensor and the object. This can only be done by using a more complex imaging sensor, increasing the complexity to define a validity function. In case of a distance sensor, the only way is to create a model of the workspace and the robot and then use this to determine if the robot occludes an object. But this is an even more difficult and time consuming task contradicting our goal of fast and simple development for non-experts.
The third case imposes the least workload onto the robot system and is relatively easy to implement. Only if the position is accessed in the database, the sensor signal is evaluated. While this is an adequate approach in most cases, it does have its limitations as illustrated in the examples above.
When the fourth option is pursued, the developer must set specific markers in the program to inform the database that a sensor shall be evaluated at this point. While this approach offers the highest flexibility it has some serious drawbacks:

- It violates the concept of strict separation of robot instructions and sensor data processing. Although sensor data processing is done in the database, the developer must access the program code to integrate sensors.
- Every time a sensor is evaluated, the database must check all positions if they are influenced by that sensor. For large databases this imposes a significant workload on the system. This applies to options 1 and 2 as well.
- When a different type of sensor is used or the program is modified, the position markers must be checked for correctness again. This reduces re-usability.

Because of this, the third option is the most generally applicable. We will show in Chapter 7 that in most cases the problem of occlusion and faulty sensor data can be avoided already by either altering the robot's movements or placing the sensor in a different location. We will also describe a modification to the developed software to explicitly evaluate a sensor signal at a given point in time without having to provide a validity function.

4.5 Applicability of the Proposed Concept

In this section we evaluate the proposed programming concept on a theoretical level. Our focus is to evaluate which applications can be solved with our approach and if there are limitations that impede a successful execution for some tasks.

In a first step, we will take another look at changes in the robot's workspace. For this we assume that we have measured the workspace and all objects before we start the execution of the task. That is, we know every position of every object at this point. Now, when task execution starts the workspace will change. At the very least because of the motions of the robot. In addition, the robot will alter the workspace somehow - this is the task it is supposed to perform. But there may be other alterations as well, e.g. objects will be delivered via a conveyor belt. So we classify changes of the workspace once more, but this time with another objective: Is the workspace altered only by the robot or by external actions as well?
If the only modifications to the workspace are made by the robot, we can maintain a very exact model of the workspace, since we know all changes because they have to be made by the robot. This is called a *static workspace*. An example for this is locating an object on a non-moving conveyor belt (see Figure 4.8a). Because the position will never change unless the robot moves the object, we can determine its position using sensors and be sure that it will remain in the measured position until the robot accesses it. Note that we still assume that no planning capabilities are required to solve the task. So in principle we can solve all manipulation tasks in static workspaces with the new concept.
If we allow other machines aside from the robot to modify the workspace, we call this a *dynamic workspace*. In this case, we must further differentiate between two types of modifications

1. We classify a modification of the workspace as *discrete* in the sense that a modification will occur at some point in time and the system describing the workspace will enter a new state and remain unchanged for a prolonged period of time.

2. We classify a modification of the workspace as *continuous* in the sense that a modification will occur constantly during parts (or even the whole) of the execution.

Discrete dynamic workspaces are of the same difficulty for the robot as static workspaces, because the same assumptions hold: If sensors are set up accordingly, the robot will detect that change and can compute a suitable modification of the corresponding position. Because this change remains constant, we can argue that the workspace has entered a static state once again. An example for this is shown in Figure 4.8b. Here, we assume that the conveyor belt transports the objects until they reach a light barrier at the end of the belt. When this happens the belt stops. While the belt is moving, the workspace is altered dynamically. But when it stops, the workspace enters a discrete state once more (until the object blocking the light barrier is removed). If the state would have altered once more before the robot could act accordingly, the workspace would be continuous dynamic.
In case of a continuous dynamic workspace we must take a closer look at the response time t_r that passes from detecting and processing a change and approaching the altered p. If in that time p only changes so much that it is still accurate enough given the tolerances of the task, we argue that all tasks belonging to this class can be regarded as discrete dynamic. This holds because the robot is fast enough to react to the change although p will alter continuously. An example for this is shown in Figure 4.8c. The conveyor belt does not stop,

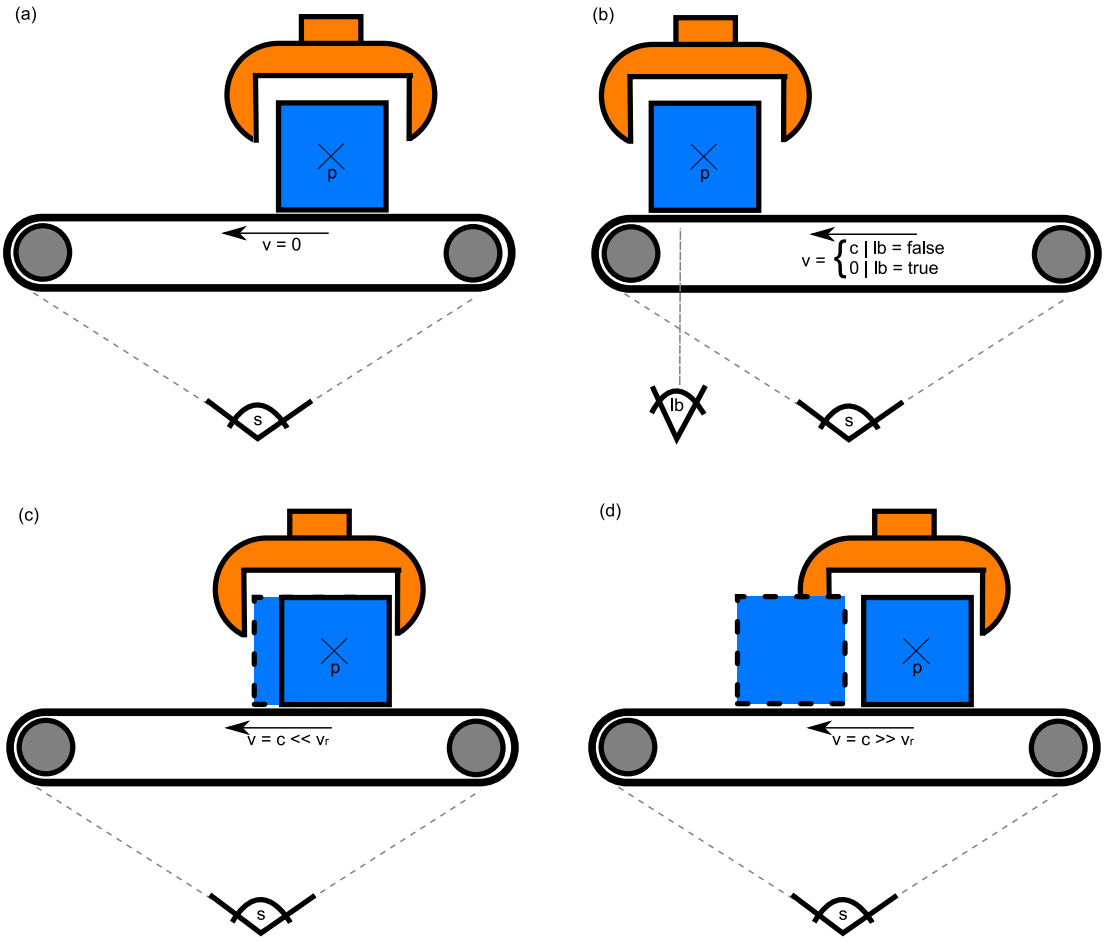


Figure 4.8: Examples of different types of environmen. In all cases the robot shall locate the box in position p on the conveyor belt using sensor s. (a) depicts a static environment. (b) depicts an discrete, continuous environment. (c) depicts a continuous environment. (d) depicts a continuous environment where the change happens too fast for the robot to react in time.

but moves at constant speed that is set so low that it will move only by an insignificant distance in t_r until the robot has moved to this position.
If t_r is high or the workspace changes relatively fast, the actual position p_a will differ significantly from the calculated p. If the speed v by which p moves is constant (and we either know this speed or can measure it somehow), we can employ *live coordinate systems* that allow us to model moving positions: We calculate p and apply a live coordinate system that moves at speed v. Now the position will always be accurate no matter at which point in time the robot will approach it. If v is not constant, then there is no way we can monitor that change and act fast enough to encounter it. An example is shown in Figure 4.8d. Here the speed of the conveyor belt is set so high that the robot is not fast enough to grasp the object because it will have moved significantly from the calculated position.
In summary, we can say that the proposed concept of adding extensions describing sensor

evaluation to robot positions allows manipulation tasks in static and dynamic workspaces unless the speed at which the monitored positions move is significantly higher than the robot speed or is not constant.

4.6 Conclusions

In this chapter, we have shown that all workspace changes can be classified by two metrics: The origin and the occurrence of the change. Changes that require external sensor for compensation are those that occur frequently, in up to every execution. In a second step we have classified optimizations of robot programs into four layers of abstraction. We have shown that for our work the main problem is to design sensor dependend positions and optimize these with respect to execution time. Based on these two findings we have outlined a position-centered approach to program robot handling tasks. With this approach, sensor data processing is coupled to the position database instead of integrating it into the robot program. The proposed concept is designed to be independent from both the task and the type of sensor used. We have evaluated at which point in time the sensor information may be processed and when the position database is to be updated. We have argued that it is sufficient to evaluate the sensor when the position is called by the robot. In a last step, we have theoretically evaluated what types of workspace changes in handling tasks can be encountered with the proposed concept. We have argued, that in principle, we can deal with all types of workspace changes, unless the alteration happens significantly faster than the robot's reaction time or is non-linear.
In Chapter 7, we conduct experiments to evaluate the proposed concept in practice. But before that we address two problems, that we have factored out so far: In Chapter 5, we deal with changes requiring iterative compensation. This will require more extensions to the framework. Another factor we have not dealt with so far is the concept of adaptivity. For preparatory sensors, there is no need for this as the robot can react immediately to the change. We will describe various methods to optimize a robot program when concurrent sensors are used in Chapter 5 as well. In Chapter 6, we deal with changes that occur unintendedly: drifts caused by abrasion and faulty alignments of objects in the workspace. Here, we can also employ adaptivity methods to predict arising drifts and encounter these.

Chapter 5

Variations

Variations are the main reason to employ external sensors in robot handling tasks. In this chapter we will explain approaches how the framework devised in Chapter 4.4 can be used to handle variations. In Section 5.1 we define the term variation and classify different types of variation by the number of corrections required to compensate them. We show that there are three different types of variation. We explain that the framework designed so far is already capable of handling one of these types. The other two types of variation must be compensated using search motions. In Section 5.2 we evaluate other approaches to conduct searches in industrial robot tasks. In Sections 5.3 and 5.4 we explain approaches to deal with the remaining two types of variation - blind and informed searches. We focus on the optimization of these searches by adaptive means in order to speed up execution time. In a separate Section 5.5 we explain how searches can be used to learn geometric change functions during task execution. We summarize this chapter in Section 5.6. The results of Section 5.3 are published in [43].

5.1 Characteristics of Variations

In Section 4.2 we have explained that a variation is a continuous change caused by the task. At this point, we will define a variation more precisely.

Definition 4 (Variation) *A variation is an alteration of either the position, the orientation or a characteristic of an object involved in a robot handling task from one execution of the task to the next. This alteration can not be forecast or predicted by mathematical models.*

Unlike a drift (see Chapter 6), a variation is caused by the task and does not have a preferred direction. How we can deal with a variation is closely related to the question if the sensor can provide the robot with enough information to compensate the variation in a single motion or if multiple motions may be required.

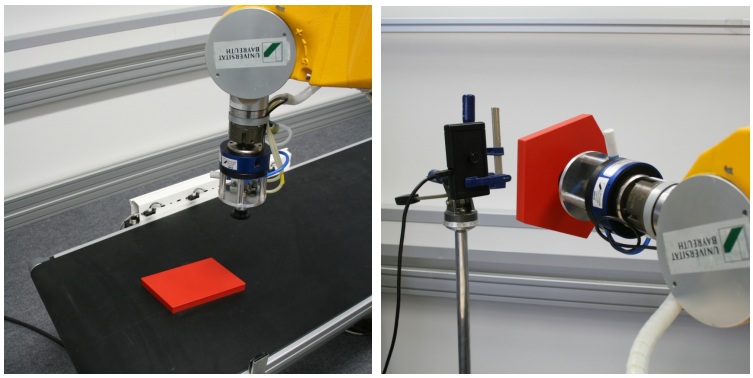


Figure 5.1: Variations that can be resolved using preparatory sensors. (a) A distance sensor is used to locate an object on a conveyor belt using a geometric property. (b) A camera is used to determine the shape of the object and place it accordingly using a conditional property.

5.1.1 Direct Compensation

In many cases, the robot can compensate the variation with a single motion. Typically this is the case if sensors are used preparatoryly. This is because we can determine the state of the object without the need for the robot to perform a series of motions[1]. The sensor will provide us with sufficient information to resolve the variation and modify subsequent motions accordingly. This is enough to compensate the variation correctly. It is not of importance if the sensor provides us with a geometric description of the variation or a conditional property or both.

- In case of a geometric property, we use the corresponding change function to calculate a Cartesian description of the variation and apply this to the default position as described in Section 4.4.1. An example for this kind of variation is shown in Figure 5.1 on the left. Here, a distance sensor determines the location of an object on the conveyor belt. The robot evaluates the corresponding change function and can immediately grasp the object with a single motion.

- In case of a conditional property, we use a mapping to link the state of the object to a set position as outlined in Section 4.4.4. An example for this kind of supervision is shown in Figure 5.1 on the right. Here, the robot grasps the object and holds it in front of a camera. A conditional property function is used to determine the shape of the object. Depending on the shape, the object is placed in a corresponding position. Again, this can be done with a single motion.

- If the sensor is capable of measuring both, a geometric property and a conditional property, we can discard the conditional property and just use the geometric property.

The framework outlined in Section 4.4 is already capable of handling variations that can be compensated with a single motion. The extensions used for this are described in Sections 4.4.1 and 4.4.4. Sequence diagrams of the algorithms are shown in Figure 4.7.

[1]This can still imply that the robot has to move to a certain position or grasp the object before the variation can be measured.

5.1.2 Iterative Compensation

Sometimes the robot cannot compensate the variation in a single motion. The robot must perform multiple measurements of the variation and corrective motions until the variation has been compensated. A typical example are insertion procedures. Here it is usually impossible to insert an object with only one motion. Instead the insertion is executed partially. Then the state of the object is measured anew and the next part of the insertion is executed. This case requires at least a condition function to evaluate if another motion has to take place or if the variation was encountered correctly. In case of an insertion this would mean, that the sensor must at least be able to check if the insertion was successful or must be continued.
If we only needed to infer a geometric property, this would imply that just a single motion is sufficient to encounter the variation. In this case we perform a supervision, which we are capable of solving already as outlined in the previous section.
For variations that must be compensated iteratively, there are two possibilities:

- There is only a conditional property available to check if the variation has been encountered correctly. In this case the robot must perform a *blind search*: The search path is fixed and does not depend on sensor information. An example for this kind of variation is shown in Figure 5.2 on the left. A force/torque sensor is used to locate the hole on a plate. The information provided by the sensor can only be used to check if the robot has found the hole or not. No information is given about its location in relation to the robot's position. The search path is fixed and will be followed by the robot until the hole has been found.

- Not only is there a (terminating) condition, but also a geometric change function. In this case the robot performs an *informed search*: The change function is used to calculate the next position in the search path. An example for this kind of variation is shown in Figure 5.2 on the right. The robot shall insert an object into its corresponding slot using a force/torque sensor. The sensor is not only used to check if the insertion was successful but also allows the measurement of moments and forces to infer the next position of the insertion.

These two types of variation cannot be solved by the framework so far. The reason is that up to now, the sensor is evaluated exactly once and its result is either added to the position stored in the database or is used to select a position from the database. In order to encounter these new types of variation, the robot must be able to check the sensor repeatedly and act accordingly. This results in a loop. The terminating condition is the conditional property to check if the search has terminated.

5.1.3 Extending the Framework to Handle Search Motions

In order to conduct searches, we must first extend our framework. We define a search in an industrial handling context as follows:

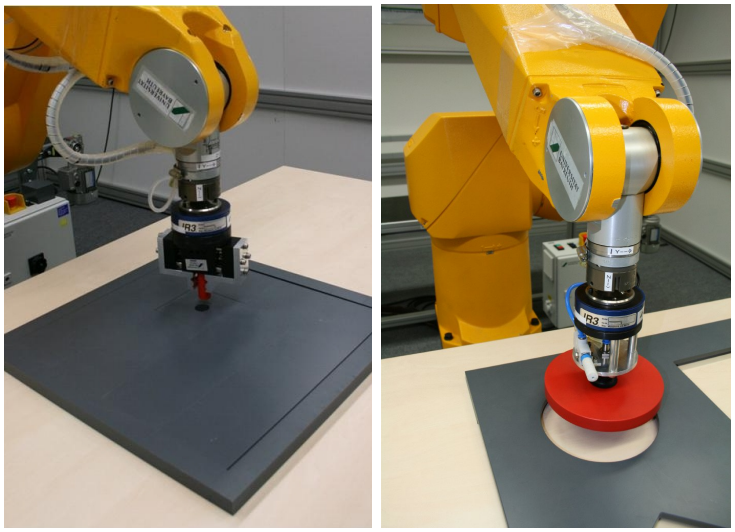


Figure 5.2: Variations that must be resolved using concurrent sensors. (a) A force/torque sensor is used to check if the robot has located the goal position while searching for a hole in a peg-in-hole task. (b) A force/torque sensor is used not only to check if an insertion was successful but also to determine the next position during that insertion.

Definition 5 (Search) *A search is the coverage of an n-dimensional region R, defined by*

- *a search path P_R covering R* ***or***
- *a function f_R evaluating some sensor signal s_v to determine the next search position*

and a condition c_R evaluating a (different) sensor s_R testing the termination of the search. The search begins in an arbitrary position in R and moves to subsequent positions within R which are either defined using P_R or by using f_R with the sensor signal of the current position. The search terminates successfully when c_R evaluates to true and fails if R has been covered completely without $c_R = 1$ in the last position.

This results in a straightforward implementation of a *search extension* for a position. This extension realizes a while-loop that terminates when c_R evaluates to true. In every execution of the loop the robot moves to the new position either specified by P_R or by f_R. The implementation in pseudocode looks like this:

Pseudocode 4 (Search extension)

```
1 WHILE(!c_R AND !searchSpaceCovered()) {
2    IF(fixed_path())
3       nextPosition = nextPositionInPath();
4    ELSE
5       nextPosition = applyChangeFunction(getSensor());
6    move nextPosition();
7 }
```

The next position in the search is either calculated by traversing the search path or evaluating s_R. The execution terminates when either c_R evaluates to true or the function `searchSpaceCovered()` indicates that R has been searched completely.
Unlike variations that are compensated directly, the robot must perform multiple motions

when processing a search. Because of this, we must augment the sequence diagram from Section 4.7. In order to conduct a search, the position database must execute a while-loop when processing the extension. In this loop the next position in the search is either extracted from the path P_R or calculated using f_R. Then this (intermediate) position is sent to the robot. When the robot has approached the position, it informs the position database. At this point the terminating condition c_R is checked.
This modification leads to a new case in the sequence diagram, which is processed for search extensions. This addition is shown in Figure 5.3.
Three things are noteworthy about this extension: Firstly, the developer does not have to deal with the creation of this while-loop, as it will be the same for all types of search. Secondly, while this construct effectively sends move commands to the robot, it is nonetheless associated with the position database and not the program code. This is because access of a position augmented with a search extension does not return a single position but an array. The robot will then move to every position in this array, but will stop when c_R is met. So the distinction between positional information and program code is maintained. Thirdly, in case of complex insertion operations it may be necessary to move from one position to the next not in a straight line but on a given trajectory. There are two ways to achieve this. Either the developer also specifies a function that computes the trajectory for two given points or the trajectory itself is split into a series of straight line motions and each intermediate point is added into the search path as well.

5.2 Related Work

Before we examine the different types of searches any further, we will (briefly) evaluate other concepts concerning industrial search motions. The topic of robots performing search otions in different areas is very wide, so we will only refer to works dealing with searches in industrial environments with results that can be transferred to this domain.
A good overview is given in [110]. Despite the fact that sensor data processing has made significant progress allowing for relatively fast processing capabilities, standard search motions which only use minimal sensory information are still commonly used in industrial applications. Also of interest are the works of [114] and [72] which cover robot motion planning based on sensor data and probability densities.
If a search cannot be avoided, usually cameras are used to supervise the search area for the given variation. While this approach is straightforward and has the advantage that the localization can be made while the robot performs some other task, this is only applicable if the search area can be monitored at all. A typical example for a task where this is impossible is the assembly of a gear box in a car. Tolerances are extremely small and the search area is occluded by other parts of the vehicle so camera supervision is impossible and local sensors must be used. Images of a key insertion using insertion maps is shown in Figure 5.15.
Sharma [109] incorporates stochastic models into gross motion planning and defines a stochastic assembly process that yields increased performance.

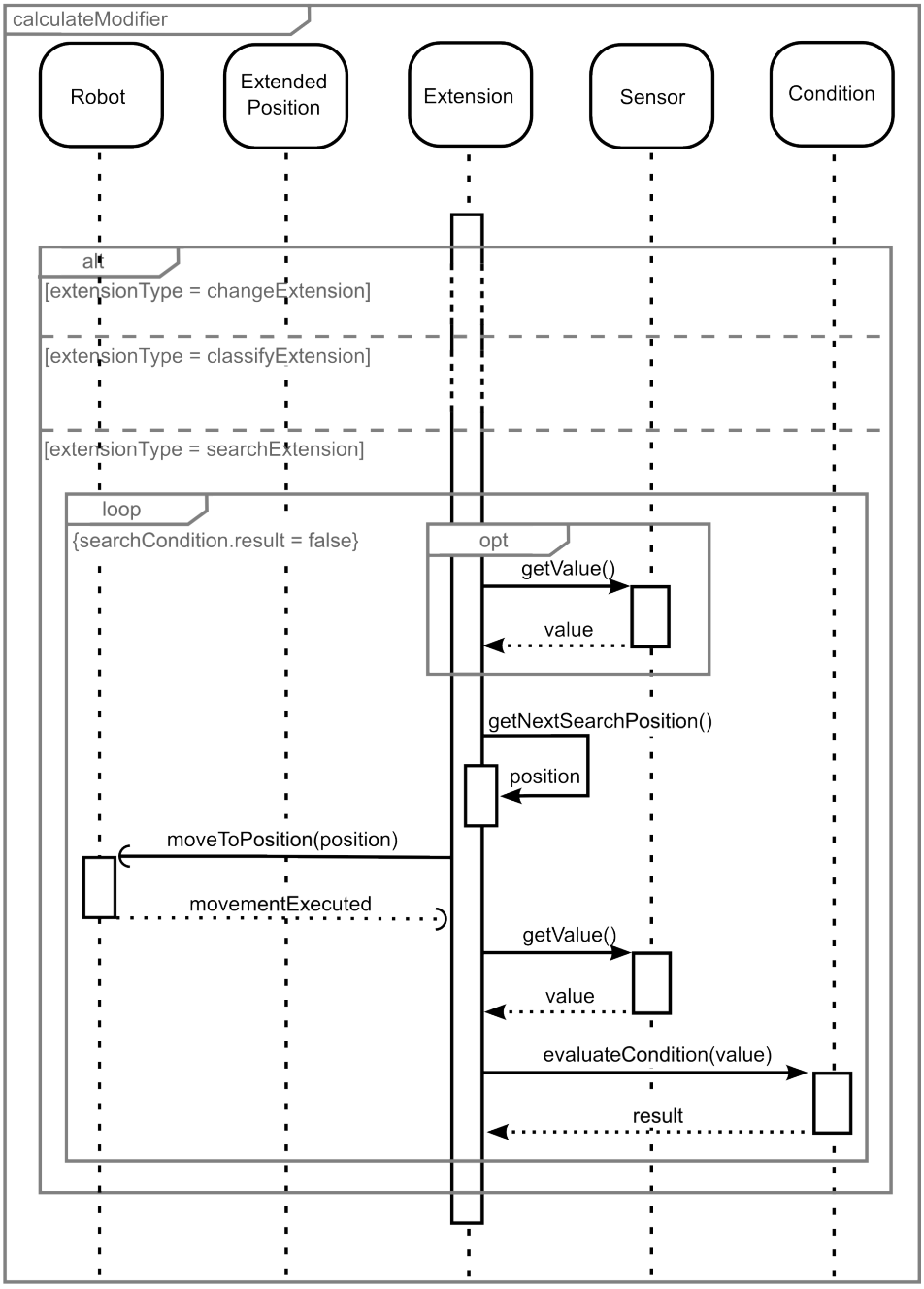


Figure 5.3: Addition to the sequence diagram 4.7 from Section 4.4.2. In case of a search motion the next position in the search is calculated (which may be done using a sensor) and send to the robot. When the robot acknowledges the position, the condition is checked and - depending on the result - the search either continues or terminates.

One important field of research outside the industrial domain is complete coverage paths in mobile robotics. Here a robot must cover an area e.g. to search for injured people after an accident. A good overview of this field of research is given by [132]. Also of interest is [61], which uses a genetic algorithm approach and knowledge gained in previous executions to optimize the path of a mobile robot.
There exist a multiplicity of approaches for various instances of the peg-in-hole problem, which is a special kind of informed search. These were already discussed in Section 2.3 and 2.4. The downside to all of these solutions is that they are geared towards specific tasks or types of sensor. There are only a couple of general approaches which will be discussed in detail in Section 5.4.
In summary, efficient search strategies are one of the central problems of robotics. While there are many specific solutions, e.g. [32] and [53], these are nearly always tailored towards specific tasks and the results can rarely be transferred to other areas.
In this work, we take a more general approach to search motions for industrial applications and outline the requirements for optimized search strategies as this widens the possibility of application. Unlike searches in unstructured environments the search area is precisely defined in industrial applications and does not change over multiple executions. The only requirement is that it provides a binary decision whether the goal of the search has been found or not.

5.3 Blind Searches

As defined in Section 5.1.3, a search is either defined by a fixed search path or a function which is used to calculate the next position in the search space. Both cases require a terminating condition. In this section, we focus on searches with fixed search paths. We call this type of search a *blind search*. The sensor is only used to check if the search has terminated. The search path itself will not be modified during the search. However, it is possible to generate different search paths for every execution. This concept will be explained in detail in Section 5.3.2.
A blind search is a motion that covers a specified area with a fixed set of motions in order to locate an object whose exact position p_g is unknown. Exactly one object is searched for at a time. Here, we set the following preconditions:

1. The search area can be m-dimensional, but its boundary in every dimension must be a straight line. The area can be divided into a set $\hat{R}$ of cells describing discrete (hyper) cubes with fixed edge length δc. When the robot moves to a cell, the whole area covered by that cube is probed. The decision whether p_g is found is binary, so there are no hints guiding us towards the goal. We assume that it takes a constant time span to check if the goal is located in a cell.

2. There are no restirctions in movements between two cells. There is no need for a neighbouring connection between two cells. No cell lying between the current and the next is tested when moving there.

3. A valid search path $P = (p_0, ..., p_{|\hat{R}|})$ must visit each cell of the whole area at least once. We include the possibility that the search fails: $p_g \notin \hat{R}$. A search path is then an ordered sequence of all cells in $\hat{R}$.

4. The distance between two cells is relevant when planning the path. There is a positive cost function $d(c_i, c_j)$ describing the time and effort to move from cell c_i to c_j. Two neighbouring cells have unit distance.

These conditions describe the general requirements imposed on a search path. Additionally, a change of direction in the search path may slow down the motion in order to perform the turn along the path. We disregard this factor here. Search paths are rated along their respective total (maximum) length l.

5.3.1 Standard Search Paths

In case a blind search is required it is impossible to specify a fixed execution time as we cannot say at which point in the search p_g is reached. Because of this, we must calculate the worst case time the search may take. This is the time it takes the robot to traverse the whole search path if p_g happens to be the very last position in the path or is not found at all. In order to maintain reasonable execution times when blind searches are employed, developers usually create search paths that always move to adjacent cells in $\hat{R}$. This approach yields the advantage that the distance between two positions in P_R is always 1. If the path has no dead ends, this will yield a total path length of $l = |\hat{R}|$. This is the optimal value for any search path in $\hat{R}$.
There are many ways to create paths with optimal length for a given search space $\hat{R}$. Mathematically, this problem is equal to finding an arrangement of all positions in $\hat{R}$ where

$$d(p_i, p_{i+1}) = 1 \ \forall p_i \in \{0, \dots |\hat{R}| - 1\} \tag{5.1}$$

In practice, there are a number of ways to order the cells in P_R algorithmically ensuring an optimal total path length. We discuss a selection of these in this section. This is by no means a complete collection, but serves to illustrate 'common practice' in modern robot programming.
For one dimension a standard search path simply orders all cells in a straight line from the beginning to the end of the search range. The search starts at one end of $\hat{R}$ and moves in steps of unit length until the other end is reached.
There are two de-facto standard search paths for searches in two-dimensional environments: A *zigzag* path and a *spiral* path (see Figure 5.4). The zigzag path starts at one corner of $\hat{R}$ and traverses along the border of one dimension until the end of that dimension is reached. Then one step is taken in the second dimension moving to the neighbouring cell. The path leads back parallel to the first line to the beginning of the first dimension. This process is repeated until the end of the second dimension is reached. In a continuous search space, the spiral path starts in the center of $\hat{R}$ and describes a circular motion which continually extends until the borders of $\hat{R}$ are reached. In a discrete search space, there are only four

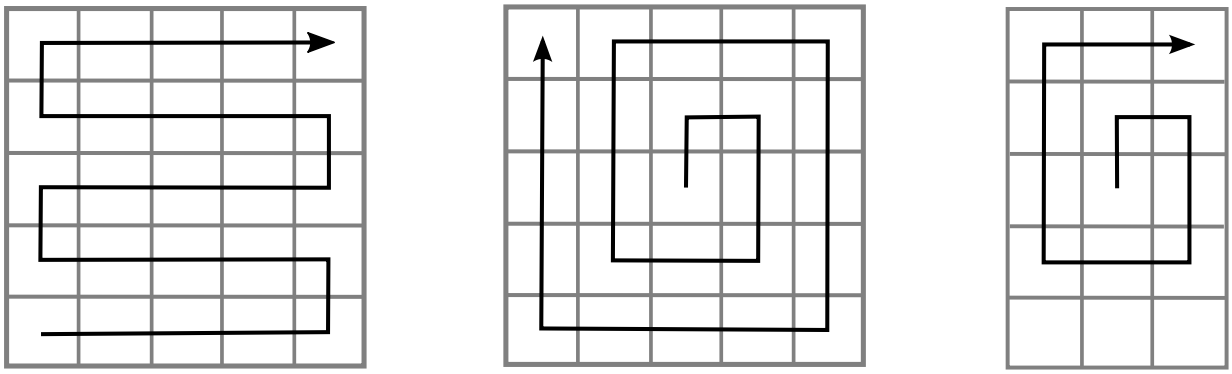


Figure 5.4: De-facto standard paths for searches in a two-dimensional plane providing a minimal path length l. Left: A zigzag path is used if there is no knowledge present about the distribution of p_g. Middle: A spiral path is used when the developer suspects that p_g is usually located in the center of R. Right: Spiral paths fail to cover R with minimal l if the search area is not square.

directions the path can take: Up, down, left and right. But the motion is analog to the continuous spiral path. Spiral paths are preferred to zigzag paths if the goal position p_g is more commonly located in the center of P_R.
Standard search paths can be generated for any dimension n using the following recursive algorithm.

Pseudocode 5 (Creation of a n-dimensional standard path)

```
 1 path createPath(int dim, int[] max_delta, int[] delta)
 2 {
 3    path P, R;
 4    if(dim == 1)
 5       return createStraightLine(0, max_delta[0], delta[0]);
 6    else
 7    {
 8       R = createPath(dim-1, max_delta, delta);
 9       liftPath(R, dim);
10       setCoordinates(R, dim, 0);
11       int delta_cnt = 0;
12       while(delta_cnt < max_delta[dim])
13       {
14          if(delta_cnt%2 == 0)
15             appendPath(P, R);
16          else
17             appendPath(P, reverse(R));
18          delta_cnt += delta[dim];
19          setCoordinates(R, dim, delta_cnt);
20       }
21    }
```

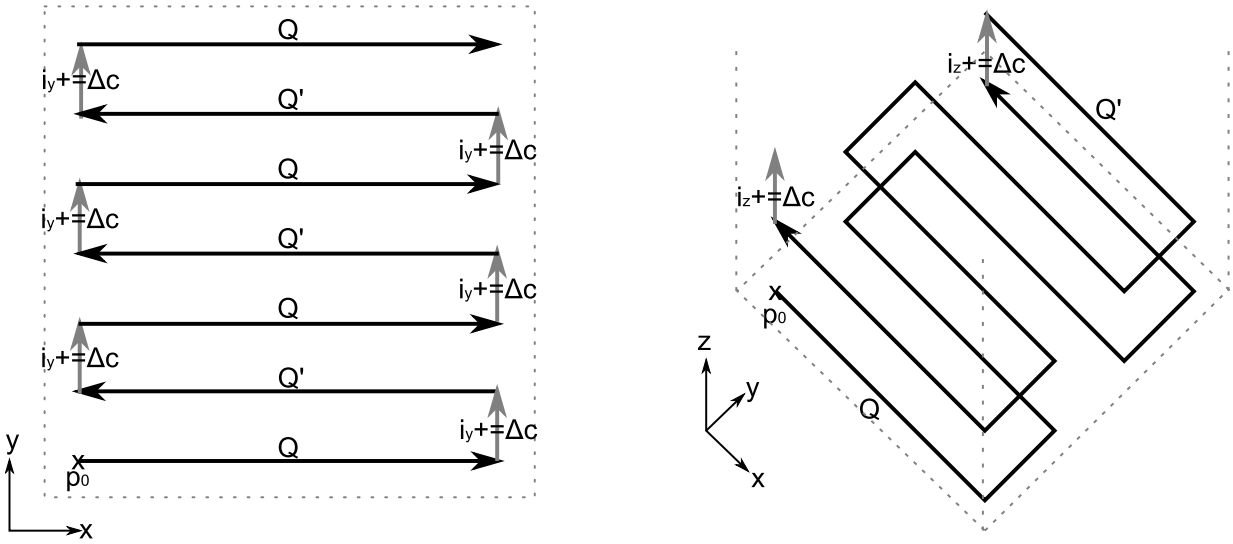


Figure 5.5: Two examples to create a higher dimensional path using a standard path from a lower dimension. Left: A one-dimensional straight search Q and its reversal Q' are used to construct a two-dimensional search path. Right: The two-dimensional search path from the left is used to create a three-dimensional search path.

```
22    return P;
23 }
```

Without loss of generality, we assume that all dimensions in $\hat{R}$ start at 0 and extend to max_i for every dimension i. Additionally, we set $\Delta c_i = 1 \ \forall i$, so we will perform steps of size one in every direction. P_R will start at $p_0 = \{m_0 = 0, m_1 = 0, ..., m_n = 0\}$.
The algorithm takes the following arguments: The dimensionality of the path and two arrays describing the extent and step sizes of every dimension. We start with an empty path $P = \{\}$ (line 3). If the dimension equals 1, a straight line is created and immediately returned. Otherwise a standard path Q of dimension $n-1$ is created. In case of $n = 1$, this will be a straight line from $m_0 = 0$ to $m_0 = max_0$ (line 5). Otherwise the algorithm calls itself to generate a path of dimension $n-1$ (line 8).
Now we have obtained a standard path Q of dimension $n-1$. We lift the dimensionality of every position in Q from $n-1$ to n (line 9). The value of the n^{th} dimension m_n is initially set to zero (line 10). We use a function `reverse` that takes a path as argument and returns the reversed path.
In the next steps, we append Q and $reverse(Q)$ in an interleaving approach to the final path P (lines 12 to 20). We append the paths in the following order: $Q, reverse(Q), \cdots$. Every time a complete path is appended to P, we increase m_n by Δc and set the dimension accordingly (lines 18 and 19). This process is repeated until $m_n = max_n$. The path P is returned (line 22).
The algorithm is illustrated for a two- and a three-dimensional path in Figure 5.5. Some aspects are noteworthy about these standard search paths:

- All paths are optimal regarding their length and no cell is visited twice. So $l = \Delta c \times |\hat{R}|$. This results in a minimal worst case time to traverse $\hat{R}$ completely.
- Spiral paths can only be generated for two-dimensional searches. It is impossible to extend such a path into three or more dimensions without losing optimality for Gaussian probability distributions.
- Spiral paths can only be generated when the number of cells in both dimensions does not differ by more than one. Otherwise $\hat{R}$ is not square and the spiral will block itself from reaching parts of $\hat{R}$ eventually (see Figure 5.4).

5.3.2 Probability Based Search Paths

The central idea of this section is to re-use knowledge gained in previous searches to create search paths tailored to the task and thus shorten the time span required for the search. Up to now, we have assumed, that we only have little or no information about p_g. In case of a two-dimensional search, a spiral search is preferred to a zigzag search if we suspect p_g to be at the center of $\hat{R}$. Apart from that we have always assumed that we have no information about the location of p_g. In this case standard paths are optimal for all dimensions of $\hat{R}$ because they are optimal with respect to the total path length.
But a robot usually executes the same task several times. Every time a blind search is conducted, we gain some information about p_g. For this, we present the concept of a search path generated along a given probability density. The idea is that the robot stores successful positions from previous executions and creates a probability density from this knowledge. This can be achieved by employing the methods described in [95]. The path is not fixed and may change with every update of the probability density.
In addition to the conditions outlined at the beginning of Section 5.3, we add a fifth precondition:

5. There is a probability density φ describing the chance that the object lies within any given cell. This density may be continuous.

Condition 5 is an addition describing the knowledge we have gained about the location of the object that we are searching so far. This allows us to start the search at the most probable cell and descend along the density instead of employing a pre-determined path. Some examples of probability distributions on a two-dimensional plane are shown in Figure 5.6.
Apart from the total path length l we will use two more criteria along which we compare different search paths to each other: (1) The expected number of cells visited $E_C(P)$

$$E_C(P) = \sum_{i=1}^{|P|} \varphi(p_i) \cdot i \tag{5.2}$$

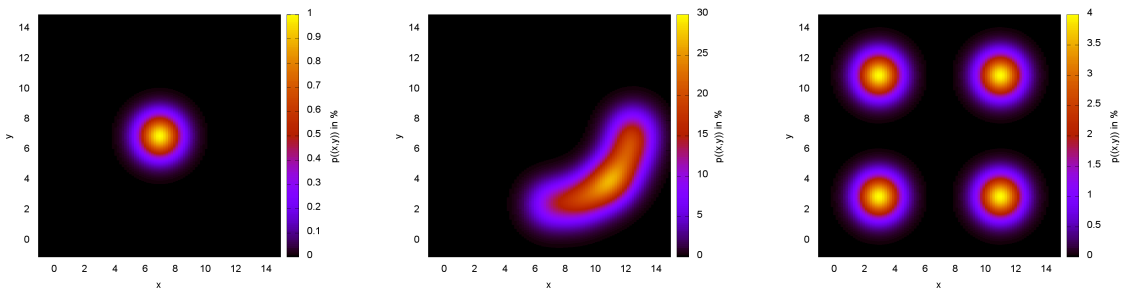


Figure 5.6: Examples for different probability densities in a two-dimensional plane. (a) A Gaussian density with center in the middle of $\hat{R}$. (b) An off-centered density where the maximum lies along a quarter circle in the third quadrant of $\hat{R}$. (c) A multi-modal density with four maxima in $\hat{R}$.

as well as (2) the expected length of the path $E_L(P)$ for the given probability density $\varphi(p_i)$ and path P_R

$$E_L(P) = \sum_{i=1}^{|P|} \varphi(p_i) \cdot d(p_{i-1}, p_i) \tag{5.3}$$

A developer faced with the task of designing a search path should consider two aspects. On the one hand, it may be useful to limit the total length to its minimal value, so the path is not exceedingly long. On the other hand if the average search time is crucial, it may make more sense to create a search path with higher total length but lower expected values. The type of search decides which of the two expected values is more important: If the movement between two cells is relatively fast compared to the time it takes to check a cell, the number of cells visited is significant. In case of slow motions, e.g. controlled movements along surfaces, the expected length is of more importance.
When standard paths are used, the zigzag path is usually chosen if the probability density is uniform, so there is no need to start at a specific cell.[2] The spiral path usually is chosen when φ is unimodal, e.g. Gaussian, with mean at the middle of $\hat{R}$. In this case the search starts in the most likely position and gradually descends along φ. Fleischer showed that this type of path is optimal if p_g is subject to a Gaussian probability distribution with a mean at the center of $\hat{R}$ [51].
Now we are interested in finding search paths that optimize the expected values for a given probability density. A search path which is minimal in this sense will find p_g as soon as possible.
To create a search path P with lower expected values than a standard path for the given dimension of $\hat{S}$, we have to approach cells with high probability first while neglecting cells with low probability until the end of the search. In case the expected length of the path

[2]All standard paths are optimal in every dimension when the probability distribution is uniform. This is because they are optimal with regard to their total length l and $\varphi(p_i) = c \; \forall p_i \in P$.

is of importance, it should be attempted to minimize huge jumps across $\hat{R}$ as much as possible. The downside is that the distance between two consecutive cells in P now may be much higher than 1. So this search path may not be minimal with respect to the total length.
In a *sorting* strategy to generate an optimized search path, the cells of $\hat{R}$ are ordered like this: The beginning of the path is the most probable cell, so

$$p_0 = \{c_i | c_i \in \hat{R} \wedge \forall c_j \in \hat{R} : \varphi(c_i) \geq \varphi(c_j)\} \tag{5.4}$$

The remaining cells of the path are chosen by a recursive definition: We always choose the next cell according to its probability in relation to its distance d to the current cell, so

$$p_{k+1} = \{c_i | c_i \in \hat{R} \setminus \{p_0, \ldots, p_k\} \wedge \forall c_j \in \hat{R} \setminus \{p_0, \ldots, p_k\} : \frac{\varphi(c_i)}{d(p_k, c_i)^n} > \frac{\varphi(c_j)}{d(p_k, c_j)^n}\} \tag{5.5}$$

The impact of the distance when choosing the next cell is controlled by the exponent n, which must not be negative. The choice of this parameter depends on the type of application and must be chosen by the developer. The lower the value of n, the smaller the impact on the distance in the selection process. So cells with a high distance to the current cell may be selected as well. The higher n is, the more the selection process favours cells that lie close to the current cell. Note that if n is set to zero, the cells are simply ordered along their respective probability. This will minimize the expected number of cells visited, but result in an extremly long total path, as it will cover great distances to move from one cell to the next (Figure 5.7, left). Vice versa, if n is set to infinity, the path always moves to neighbouring cells (Figure 5.7, right). Technically, we cannot get stuck in dead ends, because of Condition 2 (see Section 5.3). But it is possible that we must move to a cell far away from the current one, because there are no neighbouring cells left. This strategy is not heuristic but always computes the best path for the given probability density and choice of n. It may be possible that more than one path exists with the same expected value. An example is shown in Figure 5.7 on the right for a Gaussian distribution. All spiral paths that start at the center will have the same expected value regardless of the fact which neighbouring cell is visited first. The strategy presented here only computes one of these paths.h e internal ordering of cells with the same probability and relative distance to the current cell decides which this will be.

5.3.3 Experiments

In this section we describe simulation results to show how optimized search paths compare to standard search paths for various probability densities. We have limited the simulations to a two-dimensional workspace. In this case the paths are already complex compared to a one-dimensional search, but still can be visualized.

Simulation Setup

We have set up a two-dimensional squared workspace with an edge length of $15cm$. The position we are trying to find is a hole with a diameter of $1.5cm$. We have set the size of

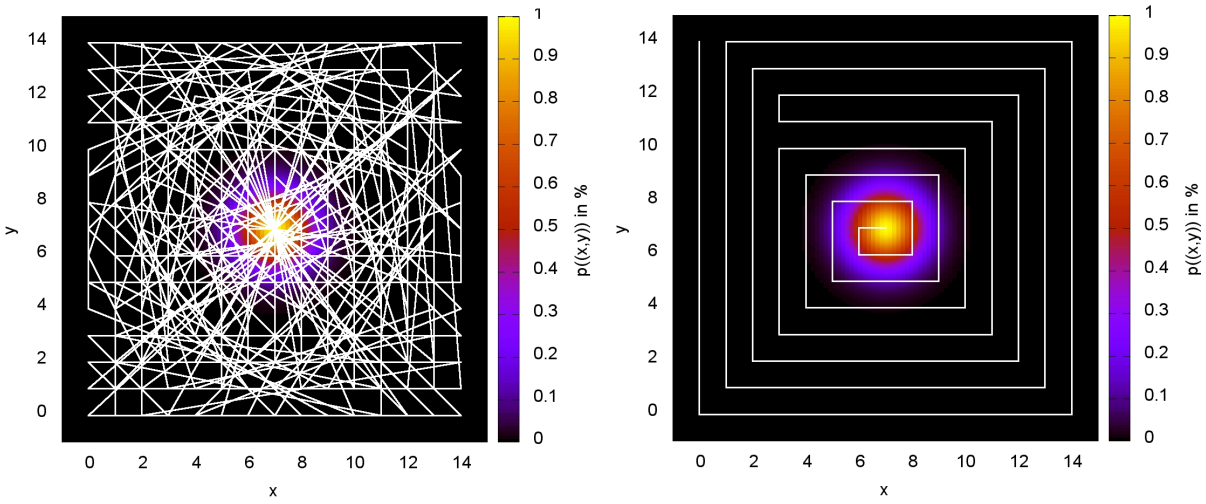


Figure 5.7: Impact of the distance when sorting cells. Both paths begin at the center. Left: The relative distance between the cells has no impact at all. Right: The relative distance between the cells is of infinite impact.

the cells in R to $\Delta c = 1cm^2$. This gives us 225 cells in the search area. So, there are 225! possible search paths, which is already too much for a brute-force computation.
We have created three probability densities:

1. A Gaussian density φ_g with mean $p_m = (7, 7)\ cm$ at the middle of R and $\sigma = 1\ cm$.

2. A mixed Gaussian density φ_m consisting of four maxima at $p_1 = (3, 3)\ cm$, $p_2 = (11, 3)\ cm$, $p_3 = (3, 11)\ cm$, $p_4 = (11, 11)\ cm$ and $\sigma = 1\ cm$ each. A typical example for such a density is a peg-in-hole task on a square plate where the hole is not centered and the plate may be rotated by 90°, 180° or 270°.

3. An off-centered density φ_c with a maximum along the third quadrant in a circle around p_m with radius $r = 5\ cm$. A typical example for such a density is a peg-in-hole task where the plate may be rotated by any value between 0° and 90°.

Plots of these three densities are shown in Figure 5.6. We have not used a uniform density, because no knowledge is present in such a density. To that effect, the expected value of all search paths is identical. The only difference will be in the total length. Because of that it is sufficient to use a standard search path.

Comparison to a Standard Path

We used these three densities and generated search paths for each one. For each density we generated paths with varying values for n in the range $[0; 10]$ with increments of 0.1. We then compared the generated paths to a standard spiral path starting at the center and measured the ratio by which the total length and the expected values differ from this

standard path. Figure 5.8 shows two paths for the Gaussian probability density for four different values of n. Figures 5.9 and 5.10 show generated paths for the same values of n for the other two probability densities.
We can see that the sorting strategy generates paths that follow the underlying density and prefer neighbouring cells. For low values of n there are significant jumps in the path. The higher n is set there more neighbouring cells are preferred. Nevertheless, these paths can lead to dead ends as well. In this case substantial jumps have to be made to reach cells not yet visited.
Now we compare this strategy to a spiral path. Figure 5.11 shows the ratio by how much the generated paths relate to the spiral path for different values of n. We compare all paths by the expected number of cells visited before p_g is found, the length of this expected path and the total path length. All results are set into relation to a spiral path starting at the center of R. A value larger than one means that the generated path performs superior to the spiral path. Vice versa a value below one means that the path performs worse than the spiral path.
In all cases, the total path length (blue line) is higher, therefore worse, than that of the spiral path, which already is optimal. But with higher values of n, the paths draw near the optimal length.
For low values of n, the sorting strategy generates paths that test significantly fewer cells than the spiral path (red line). But, because these cells are far apart, the expected length covered (green line) may be worse than that of the spiral path. With higher values for n, the two lines converge, because now neighbouring cells are preferred, similar to the spiral path. The higher n gets, the lower the advantage of the sorting strategy, because of the dominance of neighbouring cells in the selection process.

Random Probability Densities

In the next step, we have generated random probability densities in order to evaluate our strategy for a broader set of probability densities. We have created these densities by randomly placing k Gaussians uniformly in the search area. For every value of k up to 256, we have created 100 different probability densities. Then, the sorting strategy is applied to each density with various settings for n. We compare the generated paths to the standard spiral path and compute the average for each combination of k and n. The results are shown in Figure 5.12.
We can see that the generated paths perform better than the spiral path for low values of k regardless of the choice of n in terms of the expected length of the path (Figure 5.12, left). The more Gaussians are combined the more the overall probability density converges to a uniform density. In this case neither the optimized paths nor the spiral paths are superior because there is no information present in the probability density at all. When we take a look at the overall length of the search path (Figure 5.12, right), once more we can see that paths generated with a low impact of the relative distance between two cells are significantly longer than the spiral path. With increasing n, the optimized paths are nearly as short as the spiral path. There are two noteworthy aspects: The ratio increases faster

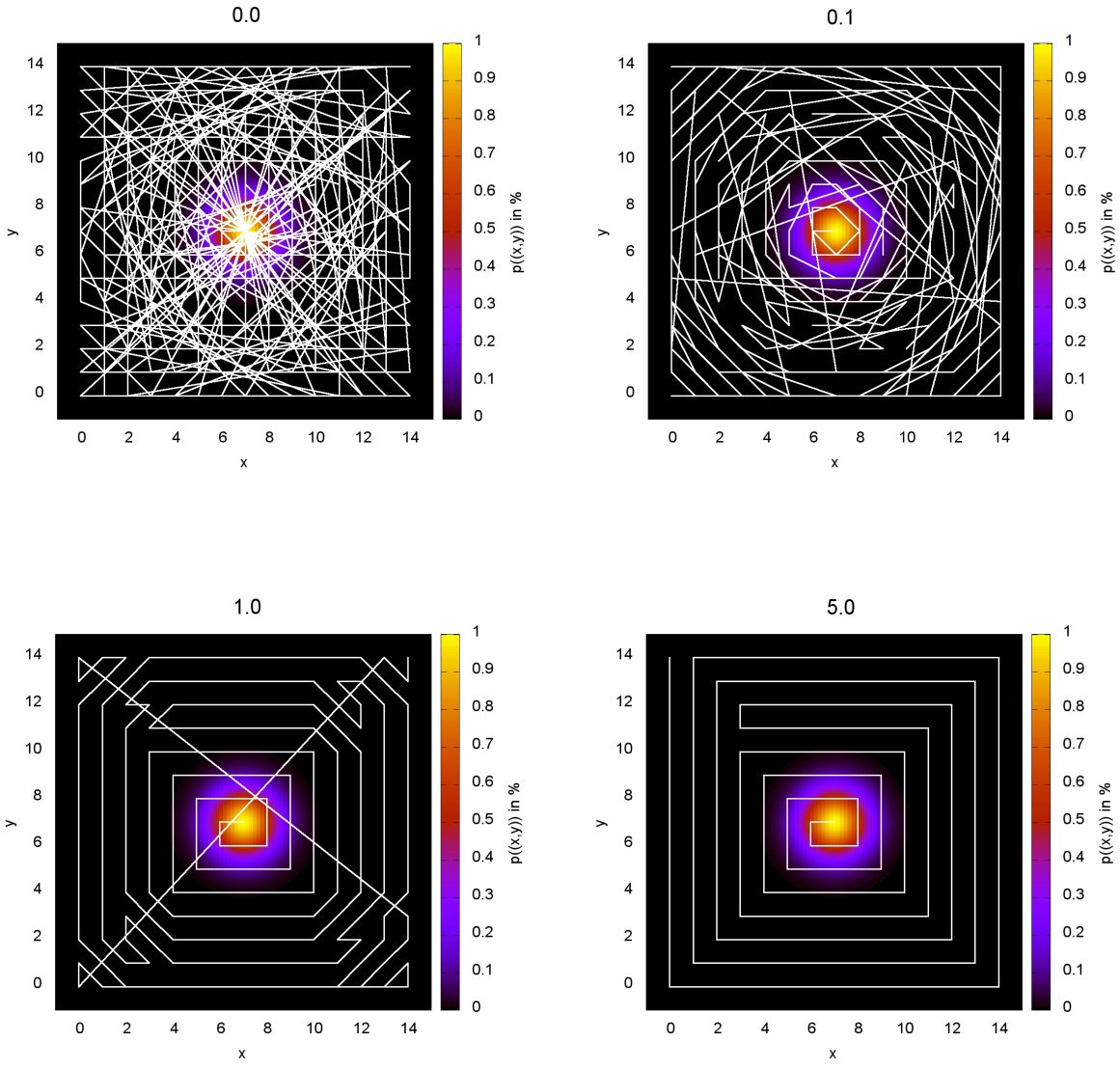


Figure 5.8: Different search paths created for a Gaussian probability density. The impact of the factor n on the distance between consecutive cells in the path is shown over each image. Top left: For $n = 0.0$ the cell with the highest overall probability is chosen next, regardless of its distance. Top right: For $n = 0.1$ there are still significant 'jumps' in the path. Bottom left: For $n = 1.0$ there are only significant jumps in the path, when the path is in a dead end. Bottom right: For $n = 5.0$ the path tries to avoid dead ends as much as possible.

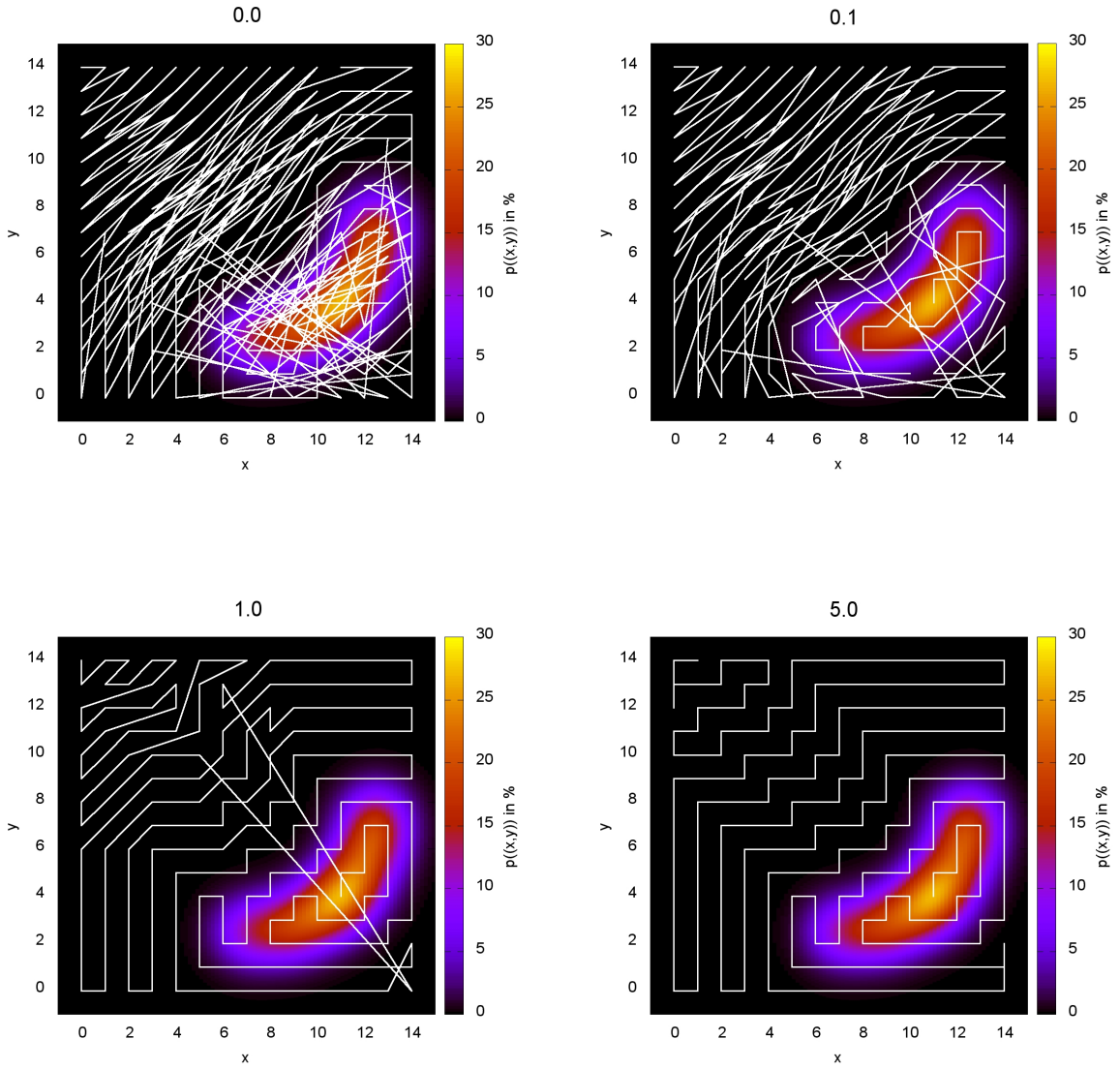


Figure 5.9: Different search paths created for an off-centered probability density. The interpretation is analog to Figure 5.8.

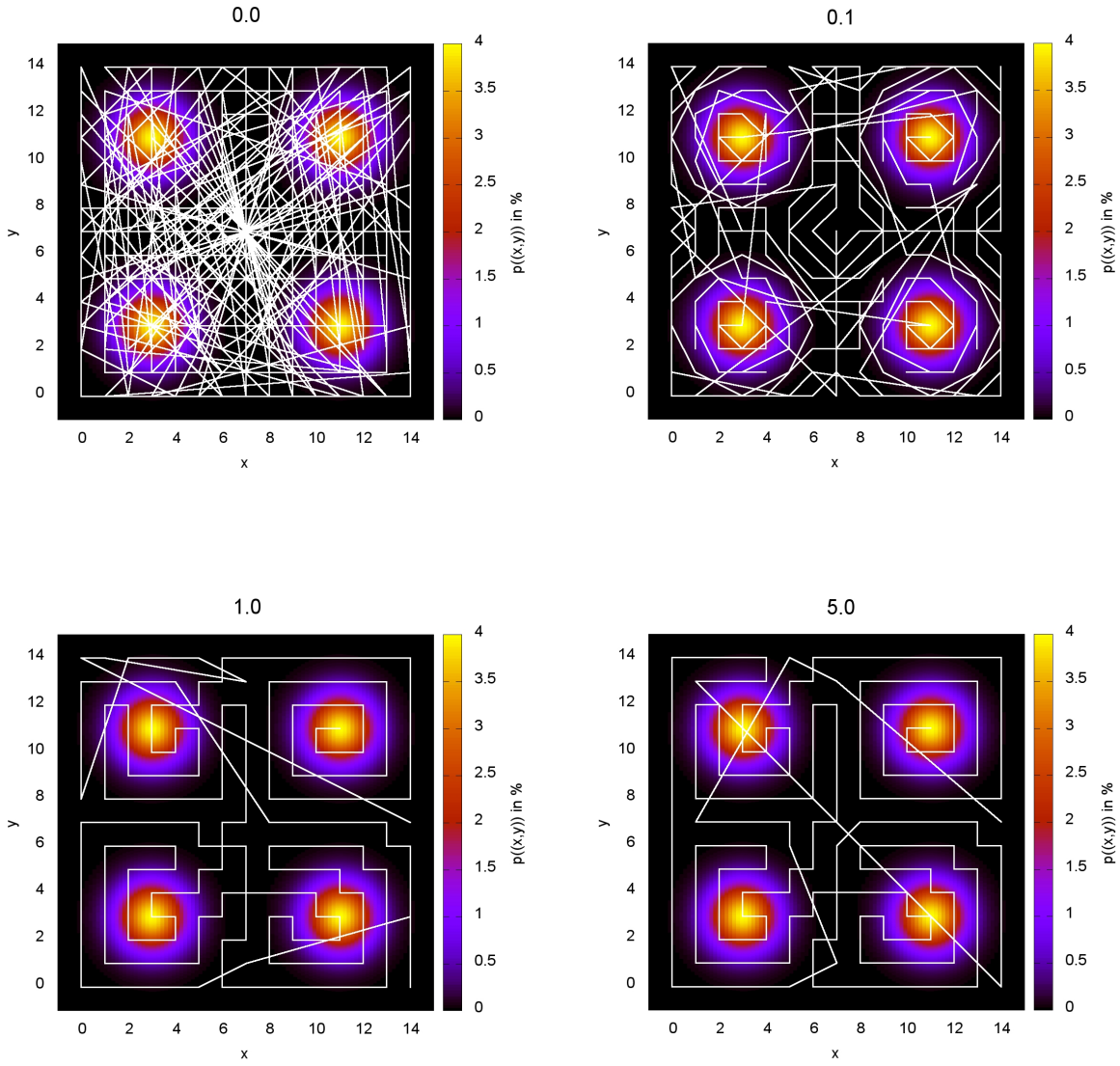


Figure 5.10: Different search paths created for a multi-modal probability density. The interpretation is analog to Figure 5.8.

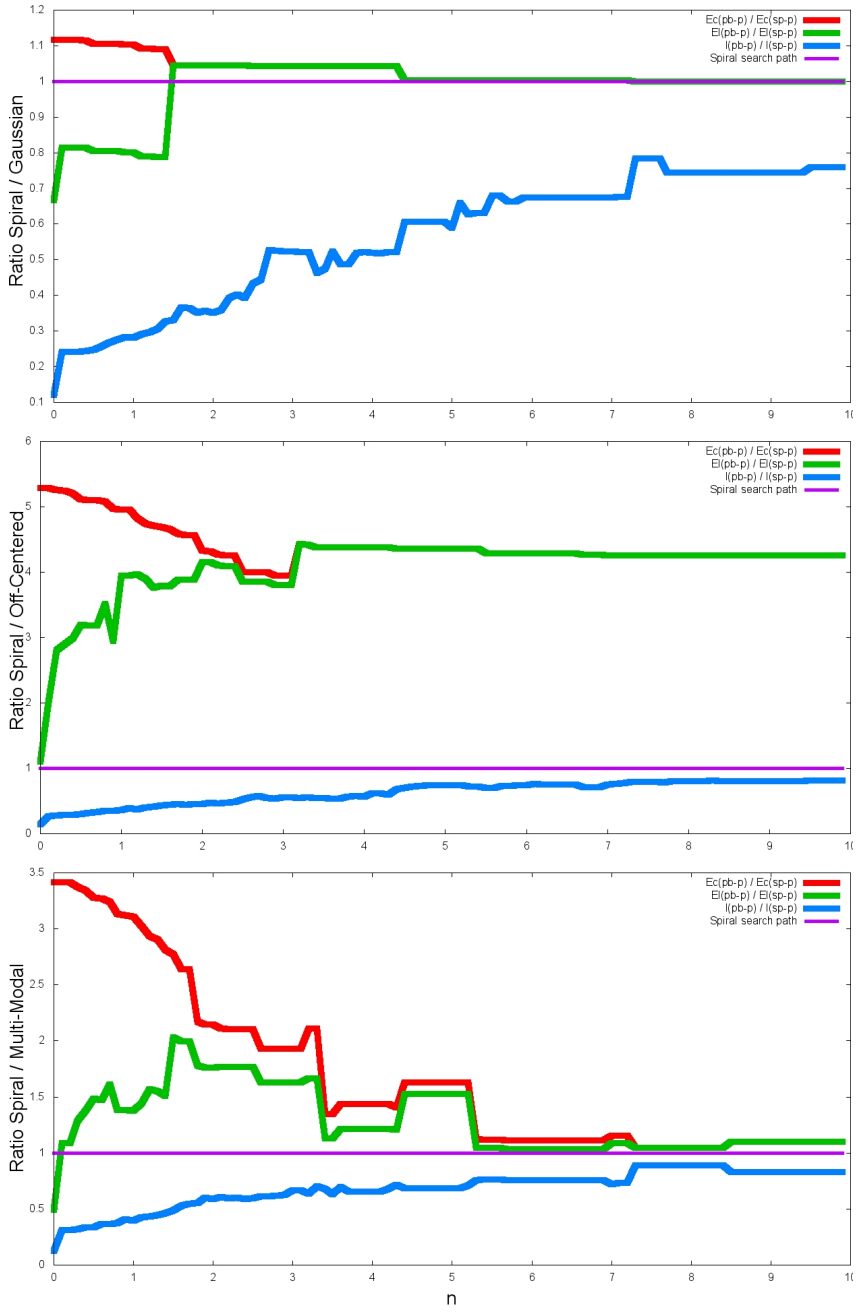


Figure 5.11: Comparison of probability based search paths to a spiral search path for the probability densities shown in Figure 5.6 for different values of n. The red line shows the ratio of the expected number of cells visited. The green line shows the ratio of the expected length of the search path. The blue line shows the ratio of the total length of the search path. Top: Results for a Gaussian probability density. Middle: Results for an off-centered probability density. Bottom: Results for a multi-modal probability density.

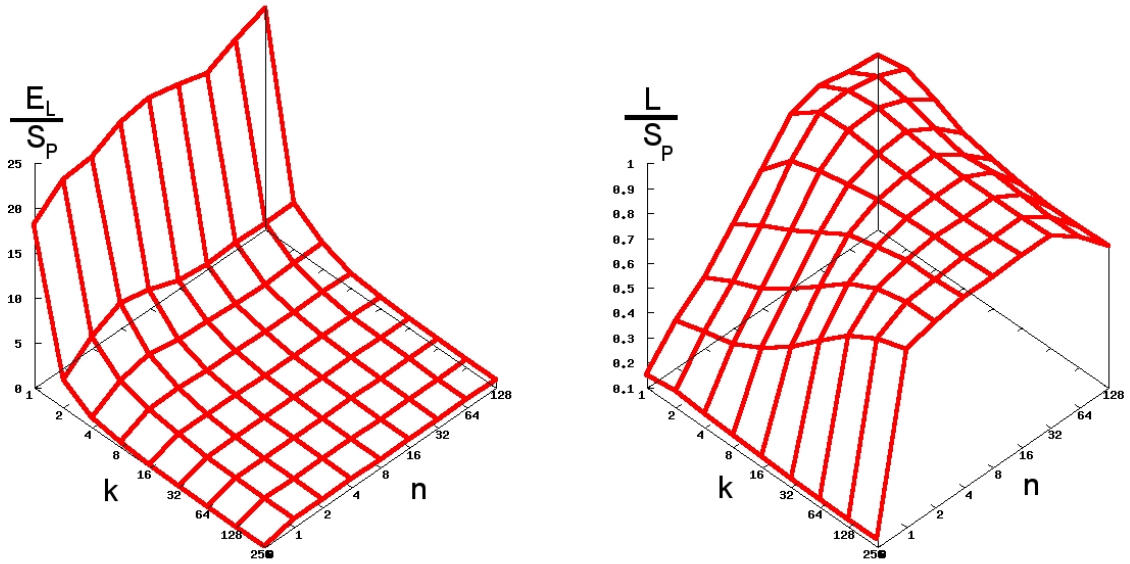


Figure 5.12: Expected length of path (left) and total length of path (right) in relation to a standard spiral path for the varying values of k and n.

for high values of k. This is because the Gaussians lie closer to each other. But for high values of k and n, the ratio decreases. This is because now there are so many Gaussians in the overall density that the path tends to lead into corners and large jumps have to be made to approach the next free cell increasing the total length of the path.

5.3.4 Conclusion

Blind searches are a part of general searches, where the search path is preset and does not change during one instance of the search.
Blind search paths based on probability densities are capable of locating the position in question faster than standard search paths. The central idea is to search in areas with high probability of success first in order to maximize the expected value.
We have described the general requirements for path planning and three ways to rate search paths. While standard paths are optimal with respect to the total length, optimized paths are improved standard paths in terms of the average time the search takes. We have shown in simulations that for Gaussian probability densities, the optimized paths perform almost as well as standard paths and even better if there is more information present about the search area. The strategy presented here is not heuristic, but always computes the best path for a given probability density and choice of the impact of the distance between two consecutive cells.
The advantage of our approach is that standard search paths can be seen as special solutions to the more general approach taken here. The algorithms to create optimized paths

are independent from the type of task. They can be incorporated into the programming environment and no additional knowledge is required by the developer. The update of the probability density describing the search area and the path planner itself are completely hidden from the developer.
This kind of search only requires a terminating condition to be specified by the developer. Depending on the task, this may be very simple or extremely difficult. As we are aiming for generality, it is not possible to describe methodologies to create such conditions. As an outlook, there is the idea of an automated analysis of sensor data recorded during (multiple) exemplary executions of the search. Since the search terminates when some aspect of the sensor data changes significantly, discontinuity detection methods [106] may be employed to create conditions automatically.

5.4 Informed Search

Blind searches constitute the simplest form of search. The robot moves along a given path until a condition is met. This approach works for a lot of tasks, but has two significant disadvantages:

1. We may gain more information from a sensor than the mere signal to terminate the search. Sometimes changes in the sensor data give us some kind of information in which direction the search should move. If the path is modified accordingly, the time required for the search may be decreased significantly.

2. Especially for peg-in-hole operations fixed paths may not make sense. Because the object can get stuck successful insertion cannot be guaranteed. Here, it is necessary to move the object in a specified sequence to achieve insertion.

To encounter these problems we establish *informed searches*. Here, in addition to the terminating condition, a function f_R is used instead of a fixed path. The function takes the current sensor signal as an argument and computes the next position of the search.
The exact type of f_R is highly dependent on the type of task and the sensor used. For example f_R for any type of search will differ significantly if a force/torque sensor or if a camera is used. Likewise, a function f_R to insert a round peg into a hole is only of limited use should the shape of the peg be square.
Because of this, the developer must create this function by himself. In this section, we outline some basic approaches of 'best practices' to create such functions. The focus lies on easy and fast creation of f_R not on optimality with respect to any of the criteria formulated in Section 5.3.
Similar to blind searches, informed searches can be conducted easily with the proposed framework. The developer not only specifies a terminating condition but also the function f_R to compute the next position in the search. It should be noted, that here f_R must be a geometric change function because the robot needs to know where the next position in the search shall be. This information can either be given in absolute coordinates or relative to

the robot's current position. The search is then realized with the algorithm embedded in the search extension described in Section 5.1.3.

5.4.1 Linear Correction

In many cases, the relation between the sensor signal and the next (or final) position in the search is linear. A classic example is the insertion of objects using a force/torque sensor as shown in Figure 5.13. An offset of the object along the x-axis will induce a moment along the y-axis. Here, the relation between the offset and the resulting moment is linear.
This leads to very simple strategy to conduct informed searches: The developer specifies which sensor data will influence the search into which direction and sets a scaling factor. This information can be encoded in a three tuple (c_i, s_j, f_j). Each tuple instructs the robot to calculate the next position p_{j+1} by taking the i-th coordinate c_i of the current position p_j and modifying it by the sensor value s_j that is scaled by f_j. More than one of these tuples can be created to allow for searches in multiple dimensions.
An example for a two-dimensional insertion of a round peg would look like this: We use a force/torque sensor which provides us with a six-dimensional measurement of the current forces and torques in x-, y- and z-direction of the robot's tooltip. The whole sensor data is encapsulated in a six-vector s, where $s[3]$ describes the current moment measured along the x-axis, m_x, and $s[4]$ describes the current moment measured along the y-axis, m_y. We create two tuples $t_1 = (0, s[4], 0.001)$ and $t_2 = (1, s[3], 0.001)$ meaning that we will modify the x-axis of the current search position by m_y scaled by 0.001 and the y-axis by m_x scaled by 0.001. The next search position p_{i+1} is then calculated as

$$p_{i+1} = p_i + \begin{pmatrix} m_y \cdot 0.001 \\ m_x \cdot 0.001 \end{pmatrix} \tag{5.6}$$

where p_i is the current position of the search.
Note that in this example no boundary checking is enforced. This means that the robot may leave $\hat{R}$. Boundary checking can be easily included in the algorithm, but may result in failing searches when the next position would be computed to lie outside $\hat{R}$. In this case, the search must terminate without success.
With this algorithm even some types of searches can be realized where the relation between sensor signal and next position is not linear but only monotonic. In this case, the scaling factor must be chosen very carefully to prevent *overshoot* for either small or large sensor values.
One idea to integrate adaptivity into this type of search is to optimize the scaling factor. When a search has been executed we can re-construct the whole search path from start to end. This allows us to compute an optimal value for the scaling factor which would have moved the robot from the starting position directly to the goal. This factor may then be used in the next execution of the search.

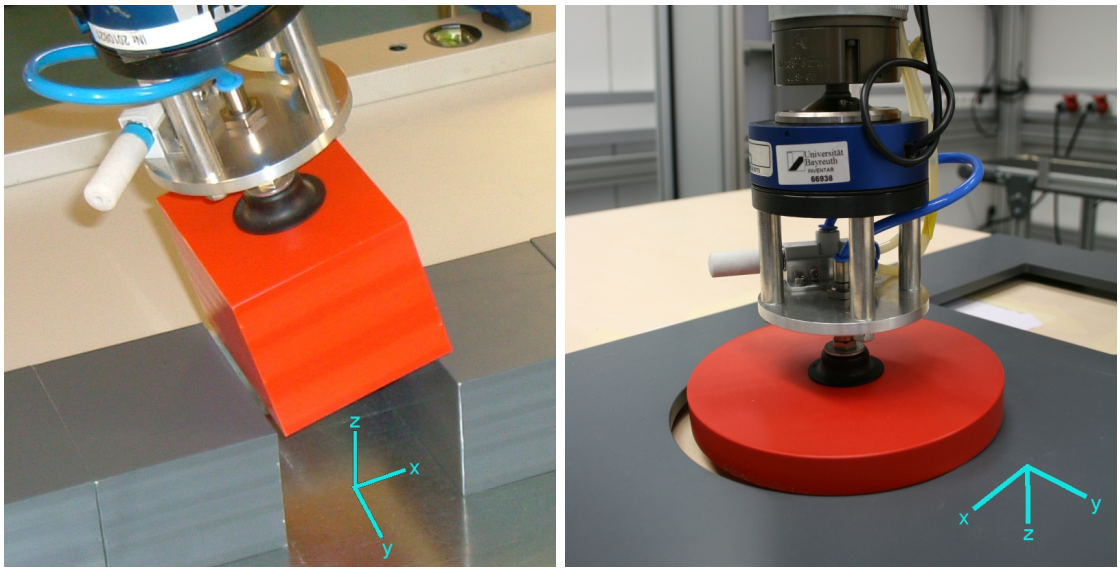


Figure 5.13: Example tasks for a simple informed search using linear correction. Left: One-dimensional insertion: A force/torque sensor is used to measure the moments along the y-axis. The measured values are scaled and applied to move the robot along the x-axis. Right: Two-dimensional insertion: The moments along the x- and y-axis are measured and scaled and applied to the y- and x-axis in order to insert the disk.

5.4.2 Partitioning the Problem

We are not trying to create an optimal solution, but favour simplicity of programming and prefer solutions which are stable enough to be successful but may be slow compared to other, more detailed approaches. One way to achieve this is to split the task into a series of smaller, more easy searches. The proposed framework allows us to create multiple extensions and add all of these to the same position.
It is up to the developer to figure out if the task can be divided into a series of smaller, simpler searches. Every search will then be realized as a separate extension.
An example is shown in Figure 5.14. We want to insert a quadratic peg into a hole. Instead of developing a complex function which inserts the peg in one fluent motion, we sub-divide the problem into three searches: Firstly, we perform a blind search to align the peg's rotation with the hole (Figure 5.14, left). We bring the peg into contact with the plate and simply rotate it until we measure a significant moment either along the x- or y-axis or if the force along the z-axis is close to zero. This will happen when the peg has the same orientation as the hole. In a second step, we use a linear correction search described in Section 5.4.1 along the robot's x-axis to align the peg with two sides of the hole. In a third and final search we perform another linear correction to insert the peg. It should be noted that this is by no means an optimal strategy and is prone to a large number of problems. More practical solutions tilt the peg in order to ensure a more stable insertion. But these strategies are also a lot more difficult to realize, especially for unskilled developers.

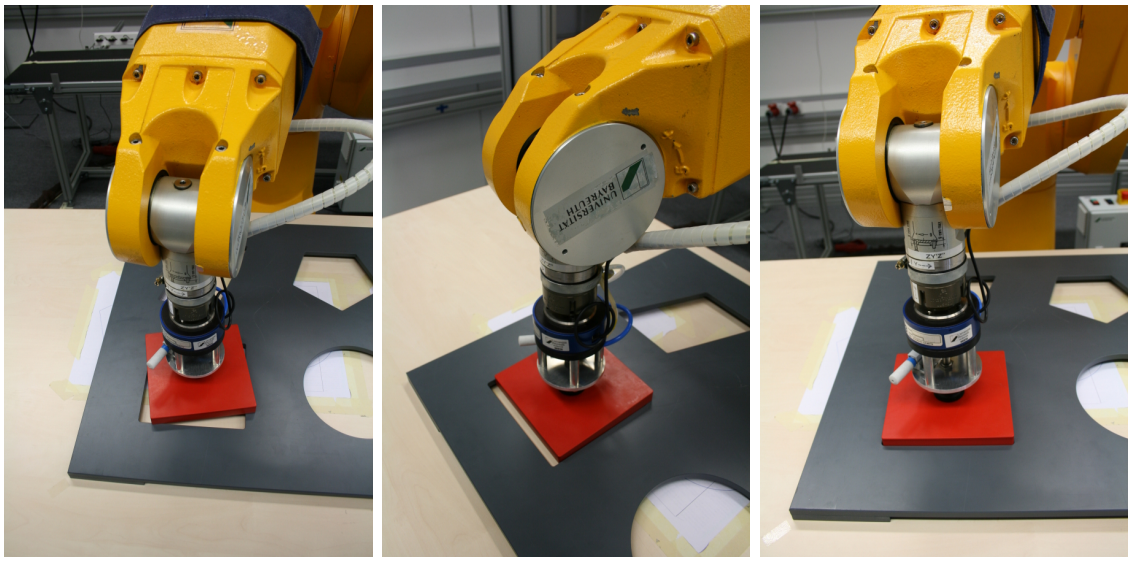


Figure 5.14: Example to partition a complex informed search into a series of simpler searches. Left: When inserting a quadratic disk, in a first step the disk's rotation is compensated. Middle: In the second step, the disk is aligned with two sides of the hole. Right: In the final step, the insertion is completed. Each step is realized using a linear correction search in one dimension each.

This strategy fails when parts of the task must be executed repeatedly. The framework only supports sequential processing of extensions. It is not possible to either skip extensions depending on conditions or repeat them for a number of times. We will discuss the implications of this constraint in Section 8.2.2. One could argue that it may be practical if *if*- and *while*-constructs were also permitted when extending a position. But it should be kept in mind that such complex extensions may also be realized with a single change-function. This change-function must employ an internal state machine to reflect the implications of previous calls. In principle this is possible, but demands high skills from the developer. In practice this results in another mixture of sensor data processing and robot motions, only that robot motions are mapped to internal states of the change function.

5.4.3 Insertion Maps

Insertion maps are a different approach to realize complex informed searches. The concept was developed by Chhatpar [33]. The central idea is to create a k-dimensional map that stores sensor values for all possible locations in the search space. The variable k specifies the dimension of the sensor signal. This is done in a pre-processing step before the actual execution and can be automated (see Figure 5.15). In execution, this map is used by an algorithm to locate the robot in R and then navigate it to the goal position. The algorithm computes a series of motions to narrow down the amount of potential locations should the current set of sensor values match multiple positions. Once there is only one possible

Figure 5.15: Example for the use of insertion maps. Left: A key shall be inserted into a lock with extremely low tolerances. Right: Sensor values for all possible locations in the search space are recorded and stored in a map. This step is performed during setup. In execution, a localization algorithm computes the matching position of the key and then traverses along the map to the goal position. Image courtesy of [33].

position left, a path is computed from the current position to the goal.
This approach can be used for complex searches and may be automated completely. The only thing to be done by the developer is to create the map. In the works of Thomas [117] this approach has been extended to compute these maps using CAD data, so that there is no need to actually perform measurements in the workspace.
Using insertion maps, complex searches can be executed easily. The downside is that this approach is only feasible if the objects involved are identical with every execution of the task. Tolerances that would alter the maps are not allowed. This poses serious problems when handling organic materials and parts which differ significantly from one another.
We have not extended this approach any further and only mention it here as an approach to allow unskilled developers to create complex informed searches. In principle this work can be integrated into our framework. The insertion map is used as a geometric change function and no other alterations have to be made.

5.4.4 Skills and Skill Libraries

In flexible robot programming, skills and skill libraries are a popular approach to encapsulate complex operations into a single command. In theory this concept may be employed here as well, as the aspect of sensor data processing remains hidden from the developer. But a skill is called explicitly in the robot program and is by no means connected to the position database. Because of this, we will not consider these approaches in this work.
Another disadvantage is that due to the large range of applications up until today, there is no universal library encompassing a set of skills capable of dealing with all situations arising in handling tasks. Inevitably, all libraries either focus on specific tasks or sensors. This forms a contradiction to the requirements of this work.

5.4.5 Conclusion

Informed searches not only use a sensor to determine if the search has terminated but also to compute the next position in the search. This allows us to perform faster and more

complex searches. This type of search is most commonly used in insertion tasks, but other tasks are also possible. The functions used to compute the next position in the search are highly dependent on the type of task and sensor used. Because of this, it is impossible to give a universal algorithm enabling unskilled developers to create such searches. In this section we have described various ways to create such functions without requiring expert knowledge from the developer.
Simple linear correction based on the current sensor data allows us to solve a large range of applications. More complex searches can be divided into a series of simpler searches. Another approach are insertion maps to handle difficult searches in high dimensional search spaces. All of these approaches are aimed towards non-expert developers and focus on fast and easy creation, but not on optimality with respect to execution time.
A general problem of every informed search is that termination of the search cannot be guaranteed. The change function that computes the next position can produce loops in the search path preventing the robot from ever reaching its goal. It is not trivial to detect these loops and terminate the search.
Universal adaptivity concepts are difficult to integrate in informed searches. So, these must be designed by the developer as well. This will probably exceed the wealth of experience of the developer. Because of this, we cannot give detailed algorithms for this kind of search.

5.5 Using Searches for Online Computation of Change Functions

In Section 3.5.2 we have outlined how geometric change functions can be learned from a scratch during the first executions of the task at hand. In this section, we show that the algorithm to do this can be encapsulated into a search extension.
The idea outlined in Section 3.5.2 was that the robot will modify the change function if it realizes the change computed from the current sensor value does not match the real position in the workspace. In this case the correct change is measured and the change function updated accordingly.

5.5.1 Using Searches to Check for Correctness

We can use a search extension to check if the position was estimated correctly and search the position if necessary. In practice we will use two extensions:

1. The first extension uses the geometric change function f_{est} and the sensor value s to modify the default position p accordingly. This is the classic approach already described in Section 4.4. If $f_{est}(s)$ is accurate enough, the robot will move to the correct position. But if it is not, the robot will move to a wrong position. If this is the case, we must determine the correct position p' and update f_{est} with a new tuple (p', s) reflecting this knowledge. But so far, we only know s not p'.

2. The second extension is a search extension. We create a condition c which checks if the robot has moved to the correct position. This condition does not necessarily need to evaluate the same sensor as the first extension. If the position was correct, the search motion will terminate directly. Otherwise the search is continued. When the search terminates, we have found p' and can update the change function of the first extension.

The update process is performed automatically. The only thing we must do is to provide a mechanism to link the two extensions, to transmit the result of the search to the first extension.
The search can either be a blind search or an informed search. Which type we can use depends on the fact whether the robot motion will modify the sensor signal of the first extension or not.

5.5.2 Blind Searches to Update Change Functions

If a motion does not change the signal, we execute a blind search. An example for this scenario is given in Section 3.6.1 where we try to locate an object using distance sensors. If the change function evaluating the sensor provides the robot with an inaccurate result it fails to grasp the rod. This is checked using the force/torque sensor mounted to the robot's wrist. The robot then searches the position of the steel rod on the conveyor belt. The rod itself will not move during the search. This means that the signal of distance sensor does not change. So we have no other option as to perform a blind search, which terminates when we touch the rod. At this point, we have determined the correct position of the object by searching and know the corresponding sensor value from the first extension. Now we can update the change function to incorporate this knowledge.

5.5.3 Informed Searches to Update Change Functions

If a motion does change the signal, we can execute an informed search. An example for this scenario is given in Section 3.6.1 where we try to correct the rotation of the grasped rod. When compensating the rotation, every motion will alter the signal of the distance sensors. We use this fact to conduct an informed search which will be faster than a blind search.
In order to use an informed search, we must specify a geometric change function to calculate the next position in the search. The secant method is a suitable algorithm to do this, which has been described in Section 3.5.2. This algorithm is independent on the type of task and sensor as long as the function is monotonic. Because of this, the algorithm can be provided by the robot system and there is no need for the developer to program this himself.

5.5.4 Updating the Change Function

In Section 3.5.2, we have argued that we must choose a representation by a data set T containing tuples (p, s) mapping a position p to a sensor value s. This set is then used to

fit the change function f_{est} to the data. This representation allows us to easily update the change function by inserting new and removing incorrect tuples. The actual implementation of f_{est} is interchangeable. It is up to the developer to determine how the tuples in T are used to approximate the change function. Any interpolation method can be employed, because no additional knowledge about the function type of f_{est} is necessary. Curve-fitting methods may be used as well, which will lead to a reasonable approximation of f_{est} after fewer executions compared to interpolation methods. But, as is the case with all adaptation and learning methods in general, the more information we have available right from the start, the faster the methods will work adequately.
We show how these extensions and the update mechanism are realized in Chapter 7.

5.5.5 Conclusion

Searches can be used to allow automatic adaptation of geometric change functions. This is achieved by adding a search extension to the basic extension. The search is only initiated, if the change function of the basic extension provided a bad estimate. Both types of search can be employed. The secant algorithm is a universal method that can be used for an informed search, allowing for a fast location of the correct position. The change function is updated automatically after the search has terminated.

5.6 Conclusions

In this chapter, we have taken a closer look at variations. We argued that variations can be classified into two categories depending on the robot's capability to resolve the variation in one motion or if more than one motion is required.
We have shown that variations that can be resolved with a single motion are already covered by the basic framework outlined in Chapter 4.4.
Variations that require more than one motion are called searches. Basic requirement for all searches is a condition allowing the robot to evaluate if the search was successful or must be continued.
Blind searches only need this condition and follow a preset search path. We have shown that adaptive probability based search paths enable the robot to execute blind searches faster than standard search paths.
Informed searches also use sensors to calculate the next position in the search path during execution. While this allows complex searches, e.g. insertions, this type of search highly depends on the task and the sensor. Because of this, we can only give some basic strategies to create change functions for informed searches. It is even more difficult to create adaptive informed search motions.
In a last section, we have shown how geometric change functions can be learned adaptively by employing search motions as a second extension. This enables the developer to refrain from creating change functions and additionally allows the change function to be updated dynamically to react to altered workspace environments.

Chapter 6

Drifts

In this chapter we address ways of dealing with workspace drifts. The problem is that the drift is unintended and not a desired property of the task. Because of that it is hard - if not impossible - to model the drift in order to adapt to it. The benefits of a suitable drift recognition and adaptation are the following: Firstly, the robot can present a warning to the supervisor of the task if a drift occurs. Additionally, a prediction can be made when the drift will have accumulated to an error. Secondly, an adaptation may be performed to adjust the robot's motions to the altered environment. Finally, the robot may be capable of performing a corrective motion to counterbalance or reset the drift. Since we must employ external sensors in order to detect a drift, we face the task of filtering the information of the sensor signal for the existence of drifts.
The aim of this chapter is to show how drift recognition and adaptation can be encapsulated into our programming framework for a robot to deal with workpiece drifts with minimal knowledge and effort by the developer.
The rest of this chapter is organized as follows: In Section 6.1 we define workpiece drifts and outline the fundamental requirements for its recognition. In Section 6.2 we give a short overview of related work in this area. In Section 6.3 we explain how drift adaptation is integrated into our programming paradigm. In Section 6.4 we describe two approaches to generate a prediction based on the data provided by the sensor. Section 6.5 describes how a typical task involving a drift can be solved with our approach and Section 6.6 summarizes this chapter. The contents of this chapter are published in [41].

6.1 Properties of Drifts

In this section, we specify the basic properties of drifts. We focus on *workpiece drifts*, that is, how a specific workpiece may change its location over multiple executions of the same task. We deal with workpiece changes only, such as position, weight, etc. We do not deal with drifts caused by the sensor itself due to temperature or lighting changes, etc.

6.1.1 Properties of Workpiece Drifts

The term *workpiece drift* describes a geometrical displacement d_t of a workpiece's position p_0 between multiple executions t of the same robot task, that is

$$p_{i+1} = p_0 + d_t \tag{6.1}$$

The difference between a drift and a variation is that the drift is characterized by a preferred direction. Variations on the other hand fluctuate around a given position. The preferred direction may not be constant and can change over time. The extent of the drift is small compared to the size of the workpiece, so a drift is usually only recognizable after multiple executions.
This definition holds for one dimension in Cartesian space. Multiple drifts in different dimensions can be combined to model more complex drifts, but for this we require that the drift in each dimension is statistically independent for all dimensions, that is

$$p(d_i|d_j) = p(d_j) \; \forall i \neq j \tag{6.2}$$

where $p(d_i)$ describes the probability of an occurring drift during the current execution.

6.1.2 Drift Recognition

In order to detect a workpiece drift during a handling task, external sensors must be employed.
The change of the workpiece's position between two executions is small and may not be detected by the sensor after only one consecutive execution. So the drift may be lower than the signal-to-noise ratio of the sensor S

$$d_t < SNR(S) \tag{6.3}$$

In order to realize this drift recognition independent from the type of sensor, we use geometric change functions. By transferring the sensor signal into a Cartesian description of the drift, we are capable of designing sensor independent methods to detect and predict a drift. To use a change function f, we must specify the default position p_0 and the corresponding sensor value s_0

$$d_i = f(s_i) - f(s_0) = f(s_i) - p_0 \tag{6.4}$$

From this point on, we will only deal with Cartesian descriptions of drifts. By measuring the workpiece's position during each execution, we build a time series $\hat{D}$ over n executions

$$\hat{D} = d_0, \cdots, d_n \tag{6.5}$$

where

$$d_0 = 0 \tag{6.6}$$

because no drift has occurred yet or the sensor has just been calibrated in the very beginning. Then, drift recognition is realized by checking $\hat{D}$ after every execution and comparing

the values against a threshold chosen depending on the SNR of the sensor. It is necessary to use a time series $\hat{D}$ because otherwise we cannot distinguish between a drift and a variation. We can only determine a drift by checking for a pattern in the workpiece's locations. Otherwise every drift would be considered to be a variation.
Using $\hat{D}$, we can determine if a drift has occurred. But, for successful adaptation to the drift, we need to make a prediction about the future motion of the workpiece. We will show how this can be achieved in Section 6.4.

6.2 Related Work

The task of monitoring a workpiece drift in industrial applications over multiple executions is mainly covered in engineering literature. Chiang deals with general fault detection in industrial applications and includes drifts in their fault scheme [34]. Kesavan and Lee have given a classification of faults in industrial systems [71]. But there appears to be no standard terminology for theses processes, with the exception of one possible example in the terminology given by Raich and Cinar in [34].
Workpiece drift occurs only in industrial tasks that are repeated multiple times. Nowadays, the use of external sensors in these tasks is still limited to applications where it is absolutely necessary. On the other hand, autonomous robots rarely execute the same task twice in an identical environment. Because of this, drift recognition and compensation strategies are usually programmed on a per-task basis and form a detailed solution for a specific drift. Sharma et al. have presented an approach to optimize robot motions for given stochastic models of an assembly process, but focus on motions and do not specifically deal with workpiece drifts [109]. LaValle focuses on robot motions as well, and takes external sensors as input signals into account, but does not make use of a drift model [77].
In summary, all of these articles are about finding a specific solution for a given problem. None of the articles use the knowledge gained in previous executions to create an adaptive drift model. Here, we are interested in outlining a general approach to drift recognition independent from the type of task and the sensors used for its supervision. Additionally, we are interested in showing how a robot can deal with a detected drift and how these strategies can be integrated into a programming environment without demanding detailed knowledge about the process from the developer.

6.3 Integration into the Programming Framework

Here, we show how workpiece drifts can be detected during execution of a robot task. Based on this we describe two ways of dealing with such a drift: *Drift adaptation* and *drift correction*. We explain how these ideas can be encapsulated into a position extension.
We will only describe how drift recognition and adaptation can be set up for a drift occurring in one Cartesian dimension. The process is similar for multi-dimensional drifts.
During setup, the developer must determine that sensor will be used to monitor a workpiece

for drifts. In the next step, the default position d_0 of the workpiece in question is recorded. This includes the corresponding default sensor value s_0. Later on, all sensor values will be compared against this value[1]. The last thing to do during setup is to specify a change function for the given workpiece and the sensor.
When this is done, the developer must decide in which way the drift shall be modelled and set the corresponding parameters for the model. We describe two possible models in Section 6.4.
In the robot program itself, the developer has three options to deal with a drift: Supervision, adaptation and correction.

6.3.1 Drift Supervision

The purpose of *drift supervision* is for the robot to check if a workpiece is moving and inform the person monitoring the task when the workpiece is about to leave the workspace. In this case the developer must specify the range of the workspace W. The current drift is extrapolated and an estimate will be formed how many more executions can be performed until the drift must be corrected.

6.3.2 Drift Adaptation

Under certain circumstances, it may be useful to alter the motions of the robot to accommodate a detected drift. We call this process *drift adaptation*. Usually, this will not be necessary because of geometric change functions. If the sensor is used preparatoryly, the robot will know the current position of the workpiece straight away and can act accordingly. If the sensor is used concurrently, the robot can modify all subsequent motions as soon as the workpiece is localized. However, there are tasks where drift adaptation is useful if concurrent sensors are used. Usually this is the case, if the Cartesian range of the change function is smaller than the allowed range of the drift. An example for a task like this is given in Section 6.5. To perform an adaptation in the i-th execution, the default position is set to the current position of the workpiece

$$d_0 = d_i \tag{6.7}$$

so the robot will use the adapted position in all subsequent executions.
After we have performed an adaptation, we can still use the drift data stored in $\hat{D}$ for drift supervision, but must accommodate the fact that drifts are described in relation to the old default position. A simple way to maintain a correct description of all drifts up to now, is to subtract all entries in $\hat{D}$ by the current total drift d_i

$$\forall d_j \in \hat{D} : d_j = d_j - d_i \tag{6.8}$$

[1]This approach requires a function to subtract instances of sensor data from each other. This is a straightforward task for sensors providing us with numeric values, but more complicated for imaging sensor data. In this case the developer must also specify a suitable subtraction method.

6.3.3 Drift Correction

Another option when dealing with a drift is to try to correct the environment or the workpiece somehow by performing an unscheduled task, which is not performed during the normal execution of the task. We call this *drift correction*. This should happen only when the workpiece is about to leave the workspace. This corrective motion can be trivial, e.g. the workpiece is grasped and moved back to d_0, but can be quite complex as well, e.g. an adjustment of a machine involved in the task. Because of this no general corrective motion can be described. This motion must be designed by the developer instead. When this motion is performed, we must reset $\hat{D}$ because we have altered the environment, so our current time series no longer represents the actual state of the environment.

6.3.4 The Drift Extension

In the cases of drift adaptation and drift correction, the decision when to perform an adaptation or correction will be triggered by thresholds d_a and d_c which are set by the developer.
All three options can be integrated into a general function `update_drift_position()`. The developer only must set the specific parameters W, t_a, t_c and define a corrective motion m_c. `update_drift_position()` is called when the drift extension is processed by the framework. The pseudocode of this function looks like this:

Pseudocode 6 (Updating Drift Information)

```
 1 function update_drift_position()
 2 {
 3    d_mom = predict_Drift();

 4    // drift supervision
 5    nr_executions = calc_Remaining_executions();
 6    notify_supervisor(nr_executions);

 7    // drift adaptation
 8    if(d_mom > t_a)
 9    {
10      d_0 = apply_drift(d_0, d_mom);
11      modify_drift_data();
12      update_prediction_model();
13    }

14    // drift correction
15    if(d_mom > t_c)
16    {
17       correction_motion();
18       reset_drift_data();
```

```
19    }
20 }
```

6.4 Drift Prediction

In this section, we describe two approaches to automatically build a model that can be used to detect the current and predict the future drift. We describe how a Kalman filter can be used for this problem and a more general approach involving ARIMA models without the need for a movement model.

6.4.1 Drift Prediction Using Kalman Filters

Kalman filters are mainly used for object tracking in mobile robots [26]. The task here is analog, so we will use a similar approach.
Because we are only dealing with Cartesian drifts, we can construct a Kalman filter that is capable of predicting the future drift independent from the type of sensor used for supervision. Here, we will use the nomenclature of [70]:
The input-vector $\hat{x}_t$ is made up from the current position of the workpiece d_t in regard to its default position d_0 as well as the current drift between two executions, which can be regarded as the current speed of the workpiece v_{d_t}. So

$$\hat{x}_t = \begin{pmatrix} d_t \\ v_{d_t} \end{pmatrix} \tag{6.9}$$

We assume that the current drift is statistically independent from the current position. Then, the covariance matrix $\sum_t$ is defined as

$$\sum_t = \begin{pmatrix} \sigma_{d_t d_t} & 0 \\ 0 & \sigma_{v_{d_t} v_{d_t}} \end{pmatrix} \tag{6.10}$$

where $\sigma_{d_t d_t}$ and $\sigma_{d_t d_t}$ describe the accuracy of the estimates.
The transition matrix A_t describes the alteration from $\hat{x}_t$ to $\hat{x}_{t+1}$ and can be computed as follows: We assume that the transition is determined by the current speed v_{d_t}, the acceleration a and the time Δt elapsed between the two measurements:

$$\begin{pmatrix} d_t \\ v_{d_t} \end{pmatrix} = \begin{pmatrix} d_{t-1} + v_{d_{t-1}} \cdot \Delta t + \frac{1}{2} a \Delta t^2 \\ v_{d_{t-1}} + a \Delta t \end{pmatrix} \tag{6.11}$$

We can rewrite this to

$$\hat{x}_t = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \hat{x}_{t-1} + \begin{pmatrix} \frac{1}{2} a \Delta t^2 \\ a \Delta t \end{pmatrix} \tag{6.12}$$

then

$$A_t = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \tag{6.13}$$

and

$$\epsilon_t = \begin{pmatrix} \frac{1}{2}a\Delta t^2 \\ a\Delta t \end{pmatrix} \tag{6.14}$$

Using ϵ_t we can compute the gain matrix R_t as

$$R_t = \frac{d\epsilon_t}{da}\sigma_a^2\frac{d\epsilon^T}{da} = \begin{pmatrix} \frac{1}{4}\Delta t^4\sigma_a^2 & \frac{1}{2}\Delta t^3\sigma_a^2 \\ \frac{1}{2}\Delta t^3\sigma_a^2 & \Delta t^2\sigma_a^2 \end{pmatrix} \tag{6.15}$$

where σ_a^2 is the variance of a. Here, Δt can be set to 1 as the time between two consecutive executions is constant, as long as the drift only occurs while the task is executed.
Finally, we need a vector z_t that describes how we perceive a drift from one position to the next. This is achieved by using the external sensor. But, if we would use the sensor signal directly, the developer would have to specify this vector for every type of sensor. By employing a change function that maps the sensor signal to Cartesian space, we can compute a general form of z_t, that is applicable for all types of sensors. In this case the perceived position is exactly the current position blurred by the SNR δ_S of the sensor, so

$$z_t = C_t\hat{x}_t + \delta_S = \begin{pmatrix} 1 & 0 \end{pmatrix}\hat{x}_t + \delta_S \tag{6.16}$$

These are all the vectors and matrices necessary for a Kalman filter to compute a prediction of the next drift. The exact calculations are described in [70] and will not be repeated here. This Kalman filter is parametrized by $\sum_t$, σ_a^2 and δ_t, that must be set by the developer. It is possible to set basic values, that work adequately well, but for optimization purposes, these should be tuned by the developer.
Using this Kalman filter we get an estimation of the current drift of the workpiece. This estimate describes how the workpiece will change its current position during the next execution.
There are two drawbacks when a Kalman filter is used for drift prediction: Firstly, we can only predict the very next drift, but no drifts in the further future. Secondly, if the drift between two executions is lower than the noise of the sensor, we cannot predict any drift at all, because the Kalman filter only uses the most recent values to update its internal state.

6.4.2 Drift Prediction Using ARIMA Models

An alternative to Kalman filters is to use an *auto-regressive integrated moving-average* (ARIMA) model. These models make use of bigger parts of $\hat{D}$ allowing for predictions of more than the very next execution. An ARIMA model is actually a combination of three models, that can be described by one parameter each: An auto-regressive (p_{ar}), an integrated (p_i) and a moving-average (p_{ma}) model.
Firstly, $\hat{D}$ is differentiated p_i times, resulting in a time series $\hat{D}'$. The calculation for the next prediction is then

$$d_{n+1} = \epsilon_t + \sum_{j=1}^{p_{ar}} p_{ar_j} d'_{n-j} + \sum_{k=1}^{p_{ma}} p_{ma_k} d_{n-k} \tag{6.17}$$

Note that - unlike the Kalman filter - no assumption about the movement of the drift (that is its velocity and acceleration) is made.
A good choice of the parameters p_{ar}, p_i and p_{ma} is usually difficult, but in this case, we can make some basic assumptions which will help us choosing suitable parameters. The weighting parameters p_{ar_j} and p_{ma_k} can be fitted automatically for given methods using [24].
The auto-regressive parameter p_{ar} must be 0, because the current drift is independent from the prediction of the last drift; otherwise the act of making a prediction would already alter the environment.
We need to differentiate $\hat{D}$ exactly once, so p_i can be set to 1. This is because we store the total drift from d_0 up to the current execution d_i in $\hat{D}$. To predict the next drifts, we are interested in the alteration from one execution to the next. So we must differentiate our time series exactly once.
The moving-average parameter p_{ma} describes how many drifts from the immediate past are used to approximate the current drift. This parameter can be chosen by the developer. The choice of p_{ma} is dependant on the SNR of the sensor and the stability of the drift. If p_{ma} is set too low, the noise of the sensor will corrupt the prediction of the current drift. If the drift changes its preferred direction relatively often, a high value for p_{ma} will take drifts into account that are no longer adequate.
Note that, if we set

$$p_{ma} = s(\hat{D}) \tag{6.18}$$

and

$$p_{ma_i} = \frac{1}{s(\hat{D})} \; \forall i \tag{6.19}$$

where $s(\hat{D})$ gives us the size of $\hat{D}$, the ARIMA model is a simple linear extrapolation using the whole time series $\hat{D}$.

6.5 Experiments

In this section we will show how all three methods of dealing with drifts from Section 6.3 can be integrated easily into a robot task using Kalman filters and ARIMA models.

6.5.1 Experimental Setup

Consider the following task: A robot places a workpiece on the entry side of a conveyor belt. The workpiece is then processed by some kind of machine. When the workpiece leaves the machine, the robot picks it up again and performs some task with it without releasing it. Afterwards this workpiece is placed onto the conveyor belt once again (see Figure 6.1). In this experiment, we will use a round disk with a size of 150 mm in diameter. The robot program for this task looks like this:

Pseudocode 7 (Pick and Place Task Containing a Drift)

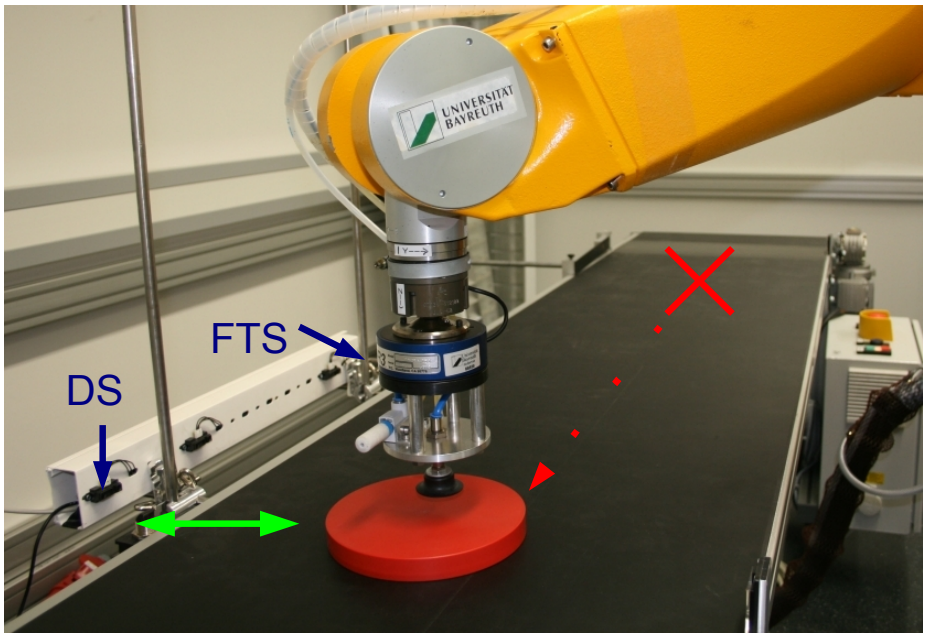


Figure 6.1: Experimental setup of the task with sensors DS and FTS described in Section 6.5.1

```
 1 repeat
 2 {
 3  MOVE p_drop
 4  RELEASE
 5  // wait for machine to finish processing
 6  WAIT
 7  MOVE TRANS_y(l_belt):p_drop
 8  GRASP
 9  // perform some other task with workpiece
10 }
```

We place the center of the robot's coordinate system into its base and the center of the conveyor belt's coordinate system into the position where the robot places the disk (see Figure 6.2). The problem with this implementation is that if the x-axes of the two coordinate systems are not exactly parallel, a drift d_y along the conveyor belt's y-axis will occur. In theory the resulting drift is calculated as

$$d_y = tan(\alpha) \cdot l_{belt} \tag{6.20}$$

where α is the angle by which the coordinate systems are rotated in relation to each other and l_{belt} is the distance between the drop-off and the pick-up position.
To supervise this drift, we use two different sensors, a force/torque sensor (FTS) and a distance sensor (DS) which are positioned in the wrist of the robot and at the pickup position, respectively (see Figure 6.1). We will try to adapt to this drift by measuring the

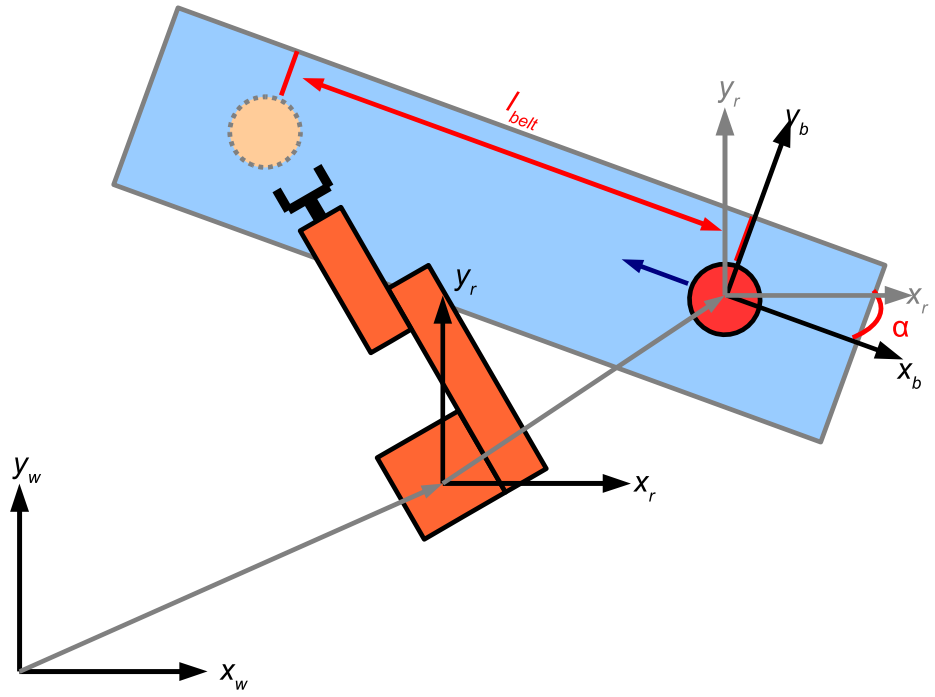


Figure 6.2: Coordinate systems of the robot and the conveyor belt in relation to the world coordinate system for the task described in Section 6.5.1

torque of the disk around the x-axis and by measuring the distance of the disk when it is to be picked up. So, the FTS is used in a concurrent way, while the DS is used preparatoryly. Here, we are only interested in checking the validity of our drift recognition algorithms, so we will not use the whole framework designed in Chapter 4 at this point. Instead we will call `update_drift_position()` from pseudocode 6 directly by adding a line calling the function before moving the robot to the pickup position in line 7 of pseudocode 7.

6.5.2 Parametrization During Setup

During setup, we record the position and the corresponding sensor values of the disk when it leaves the machine for the very first time. This defines our default position d_0 with the sensor values fts_0 and ds_0.We use the approaches described in Section 3.5.1 to determine appropriate change functions f_{fts} and f_{ds} for both sensors and the SNR of the sensors SNR_{fts} and SNR_{ds}. These are the parameters of the Kalman filters. For ARIMA prediction, we set

$$p_{ma} = \frac{1}{10} s(\hat{D}) \tag{6.21}$$

assuming that the drift will remain constant.
To parametrize the recognition and adaptation process, we measure the width of the conveyor belt w_{belt} and divide it by two, because the ideal position of the disk will be in the middle of the belt. This value will be used for drift supervision.

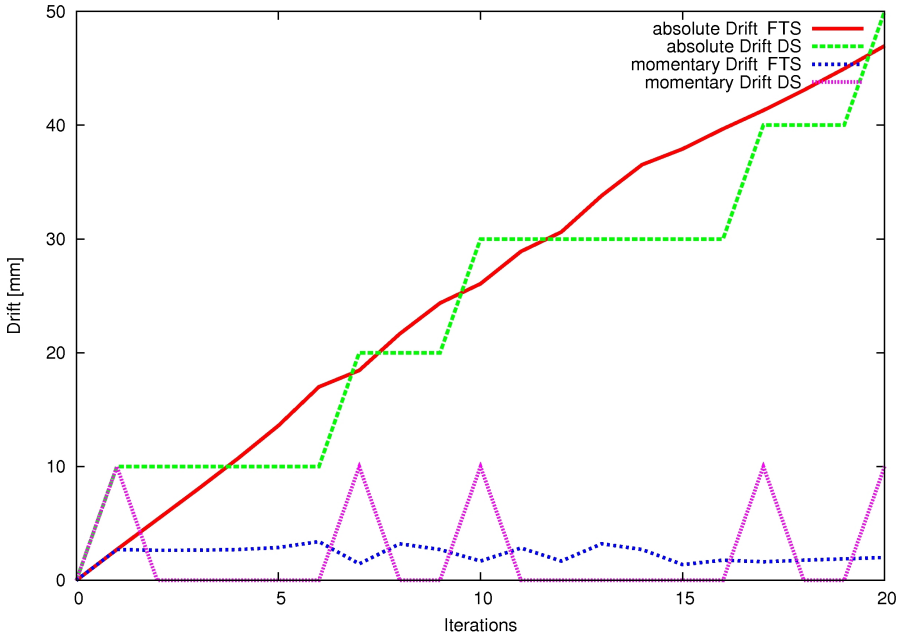


Figure 6.3: Recorded drift (absolute and current) during 20 executions of the task.

To adapt to the drift, we set d_a to half of the disk's width. So when the drift exceeds this value, the robot will modify the pickup position accordingly.
Finally, when we get too close to the edge of the conveyor belt, the robot shall pickup the disk and perform a corrective motion to position it in the middle of the belt once more. So, we set

$$d_c = \frac{2}{3} \cdot \frac{w_{belt}}{2} \tag{6.22}$$

and define a corresponding correction motion m_c.

6.5.3 Drift Recognition

We have executed the task 20 times and recorded the sensor values during each execution (see Figure 6.3), describing the total drift of the disk. The resolution of the distance sensor is relatively low, so we can only measure distances in sizes of 1 cm.
In Figures 6.4 top and bottom we have plotted the predicted drift for the next execution, the actual drift as measured by the sensor and the accuracy of the prediction for each combination of the sensor and the prediction method. We can see that after 20 executions, the combination of a Kalman filter and a force/torque sensor provides the most accurate predictions. But, in general the Kalman filter tends to oscillate while ARIMA models take longer to adapt to changes in the drift.
If a distance sensor is used to monitor the task, the predictions of the Kalman filter are worse than those of the ARIMA method. This is because the Kalman filter starts to oscillate when measurable drifts occur rarely. The ARIMA method on the other hand adapts

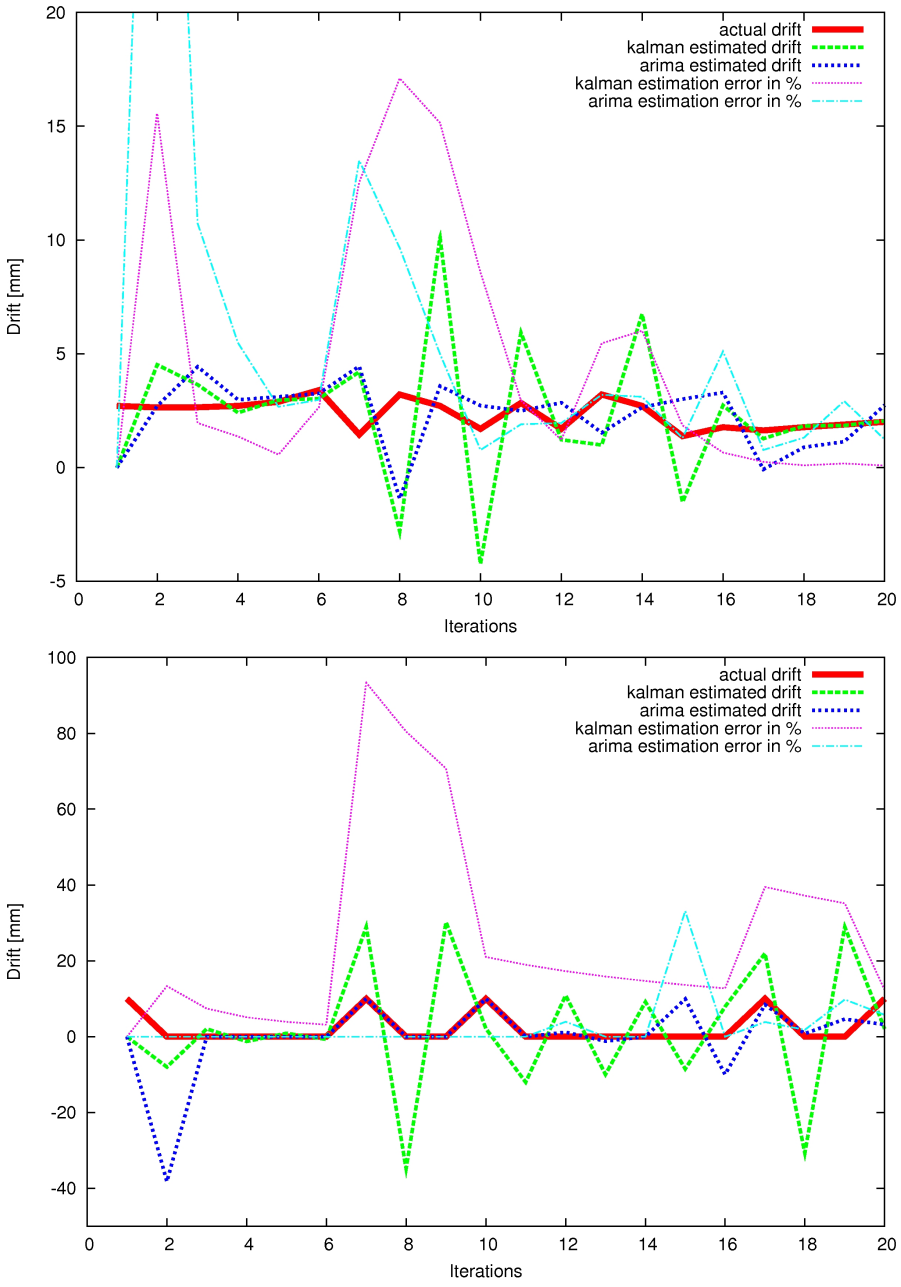


Figure 6.4: Actual and predicted drift with estimation error for Kalman and Arima models using a (top) force/torque sensor, (bottom) distance sensor.

		m	s
Real drift	FTS	2.35	0.65
	DS	2.5	4.44
Kalman prediction	FTS	2.35	3.23
	DS	1.97	17.48
ARIMA prediction	FTS	2.32	1.52
	DS	0.05	10.17

Table 6.1: Mean m in mm and standard deviation s of occurring and predicted drift for both types of sensor

relatively fast to this type of drifts.
If a force/torque sensor is used, it is the other way round. This sensor is more accurate, recognizing drifts in every execution. Because of this, the Kalman filter adapts faster, but the difference to the ARIMA prediction is less significant.
We have summarized the results in Table 6.1 showing the mean and standard deviation of the drift as measured by the sensors and as predicted by a Kalman filter and an ARIMA interpolation respectively. In general we can say that the SNR of the sensor is more important than the method chosen for the prediction.

6.5.4 Drift Adaptation and Correction

After setting up the prediction models and evaluating the drift, we set the thresholds t_a and t_c to 20 mm and 50 mm, respectively. We execute the task 100 times, measured the real drift using a Kalman filter and the force/torque sensor and log all adaptations and corrections (see Figure 6.5). We can see that the robot is now capable to keep the current drift below the adaptation threshold by modifying the pickup position according to the prediction approximately every 10 executions. To prevent the disk from falling off the edge of the conveyor belt, the robot automatically re-centers it in the middle of the belt approximately every 20 iterations.
In summary, although there is a significant drift inherent in this task, in theory we can execute this task infinitely. The robot automatically detects the drift and adapts its motion to the shifting position of the disk, as well as performing a corrective motion from time to time to reset the disk to the center of the conveyor belt.

6.5.5 Summary

Further work needs to be done in finding ways to automatically determine reasonable parameters for the Kalman filter and the ARIMA model. Simple estimates work well, but by tuning these parameters the prediction process might be optimized.
If an ARIMA model is used, the choice of p_{ma} can be optimized along the following idea: We increment p_{ma} until a significant change in the current drift is encountered. This can be

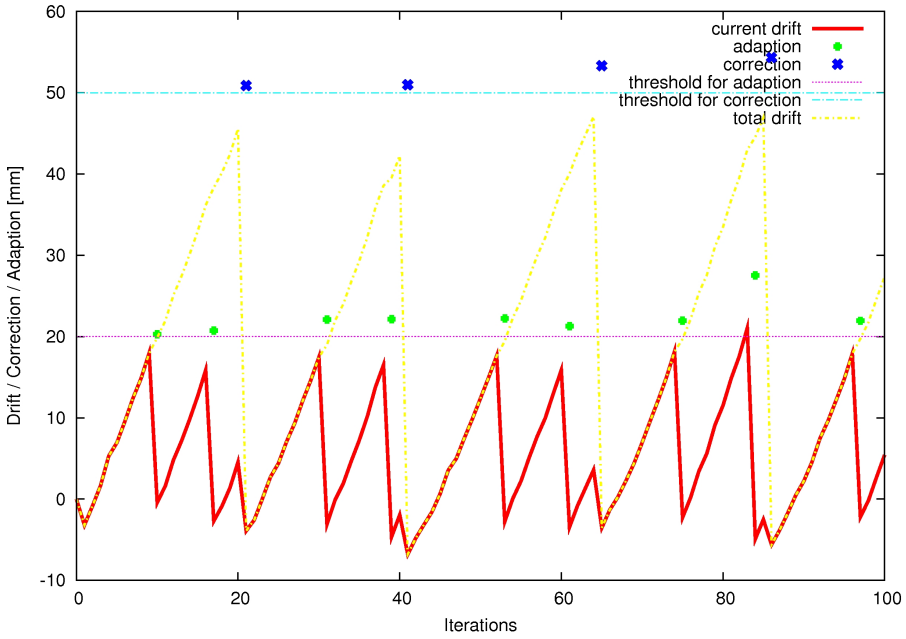


Figure 6.5: Current (red) and absolute (yellow) drift for 100 executions of the task. When the prediction of the drift exceeds the set thresholds for adaptation (pink) or correction (light blue), the robot either modifies the pickup position (green dots) or re-centers the disk (blue dots).

done using the methods described in [106]. If this happens, p_{ma} is set back to a default size. So the ARIMA model will only use values which are significant for the next prediction. We will describe how the drift extension is realized in our framework in Section 7.3.2.

6.6 Conclusions

The focus of this chapter is to show that workpieces can be monitored automatically for drifts that occur due to an imprecise setup of the workspace or an abrasion. This can be done without the need for intricate calculations by the developer. We have defined the term workpiece drift and have described two methods to detect and predict a drift for a specific workpiece and a given sensor by examining a time series describing the workpiece position over multiple executions. The presented requirements and methods are independent from the type of sensor. We have shown that these methods can be integrated into our programming framework, so that the developer only has to specify basic parameters. Finally, we have presented an experiment to validate our findings. We have shown that it is possible to employ the proposed methods to successfully detect and adapt to a workpiece drift during an automation task.
The prediction methods may be integrated into a position extension, hiding the complete drift supervision algorithm from the developer.

Chapter 7

Implementation

In this chapter, we present an implementation of the framework presented in the previous chapters. In Section 7.1 we discuss the available hardware and necessary modifications and additions to the basic concept presented in the previous chapters; we had to create a distributed system separating the robot system from the positional database. We explain the software modifications to the existing robot programming language in Section 7.2. The organization of the position database is discussed in Section 7.3. The implementation of a graphical user interface to further alleviate programming for developers is discussed in a separate Section 7.4. We test our software with a series of programming tasks of typical problems. The experiments and their results are discussed in Section 7.5. We give a short summary of this chapter in Section 7.6.

7.1 Hardware

The proposed concept has been implemented on a Stï¿½ubli RX130 with a control by Adept (see Figure 1.1). The version of the operating system is 12.3 with a corresponding version of the programming language V+. The system runs on a 86xxx processor with a system memory of 32 MB.
The programming language only provides commands to access and retrieve data from a force/torque sensor. This sensor is mounted to the robot's wrist and is manufactured by IR3. In addition, V+ encompasses a series of commands to alter and stop motions based on data received by this sensor. This is performed directly within the central execution loop of the control which runs at 60 Hertz. Motions can be altered with this frequency. Inputs at signal level are the only other information which can be accessed. So no other sensors can be accessed directly within the central control loop, but network connections and data transmission are possible. Because of this, all other sensors must be accessed on other computers and instructed to send their data via TCP/IP to the robot system for evaluation.
Another downside is that the present version of V+ only allows for variable identifiers with a maximum length of 15 characters. Variables may either be floating point numbers, strings

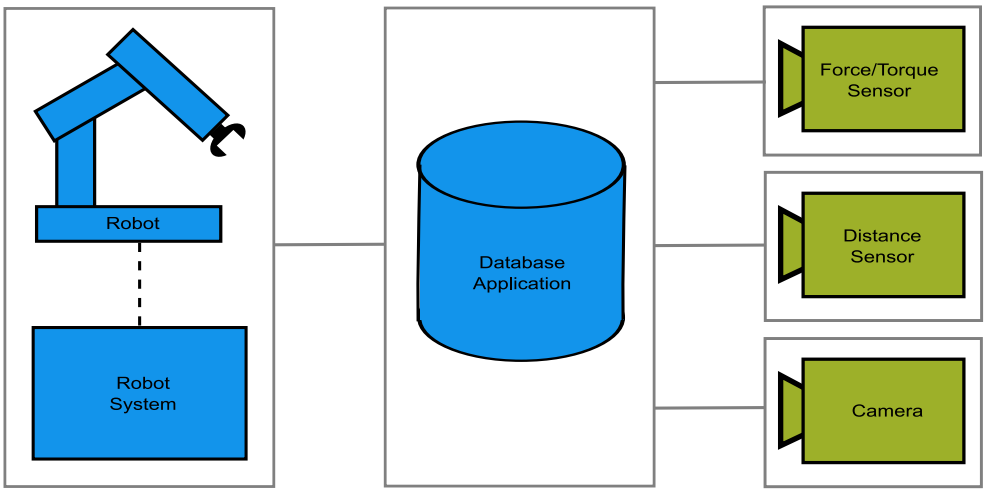


Figure 7.1: Design of the implemented system. The robot system, the position database and each sensor are realized as separate applications running on different computers (grey boxes). Communication between the components is realized using TCP/IP (grey connecting lines).

or positions / coordinate systems (called robot task frames). Complex data structures are not allowed.
Because of these limitations we have decided to realize the position database on a separate computer. Each sensor is accessed by an autonomous program and may be physically connected to a different computer. The information of every sensor is transmitted to the computer running the database. This is where all sensor data processing takes place. The additional separation of sensor data retrieval and processing allows us to easily integrate different sensors into the system. All data is sent as binary information to the database where it will be transformed by sensor transformation functions. The general concept of the distributed system is shown in Figure 7.1.
The robot requests a position using a network connection to the database and performs the movement as instructed. In case of a search this may be more than one motion.
In this system, information from the force/torque sensor will not be processed by the robot control any longer, but sent to the database instead. The downside is that motions cannot be altered within the central execution loop anymore. But this is only of minor importance for industrial handling tasks and not a general problem of the framework but caused by the hardware limitations. We will show in Section 7.5 how such motions are realized using our framework.
Processing speed of sensor information is only of minor importance in this implementation. This is because no time-critical tasks such as online modifications of trajectories are necessary. The execution will be significantly faster in a system which encapsulates all components because a large part of the response time is caused by network delays.
We are using three types of sensor:

- A force/torque sensor mounted to the robot's wrist.

- A camera which can be placed anywhere in the workspace and records RGB images.
- A series of three one-dimensional distance sensors. These are placed parallel to the side of a conveyor belt in the workspace.

7.2 Robot Software

Moving the robot is realized with two commands in V+. Both take a coordinate system `c` as a parameter: The `move` command instructs the robot to move to `c` in joint space. The `moves` command performs a straight line motion in Cartesian space. The coordinate system `c` can be constructed by concatenating multiple coordinate systems using the `:-operator`. For example, the command `move tool:trans(0,0,100)` causes the robot to move 100 mm in positive direction of the robot's tooltip, where `tool` denotes the tool coordinate system of the robot. The `trans` command takes either three or six parameters and returns a coordinate system where either the position or the position and orientation are set according to the parameters.
We have imitated the syntax of this command and created a new command called `vmove`. This command takes three parameters:

- A reference coordinate system `c_r`.
- The `NAME` of the position which is to be looked up in the position database.
- A subsequent coordinate system `c_s`.

The purpose of `c_r` and `c_s` is to allow the programmer to define positions in the database in relation to given coordinate systems (`c_r`) and allow further modifications (`c_s`) of the position which cannot be reflected by the position in the database alone. For example, the command `vmove(tool,NAME,trans(0,0,100))` instructs the robot to request position `NAME` from the database. The database evaluates all extensions connected to that position and returns the modified position. Internally, this position is then approached with the command `move tool:NAME:trans(0,0,100)`.
When requesting a position from the database, not only the position is returned but also additional information. These are flags indicating a drift or a correction of a drift. Using a second command `getPositionFlags`, the developer can check for these flags in the program. This is important if drift correction functions shall be called by the robot. The developer can insert an if-command checking the position in question for the existence of said flag and toggle a correction function.
In case of search motions, the first position in the search is sent to the robot with an additional indicator that more positions may follow. The robot will repeatedly move to the next position and inform the database to check the terminating condition. Only when the search has succeeded (or terminated with an error), the `vmove` command returns.
Sensor data processing is performed when the position is requested by the robot. As outlined in Section 4.4.5 this concept fails if the robot occludes the object from the sensor. In

this case we have implemented an additional command `preProcess`. The command takes a position name as parameter. The position is only evaluated and returned to the robot system, but not approached. The developer can store the position and approach it using the `move(s)` command at a later point. Note that this command is only sensible if no search motions are attached to the position.

7.3 Position Database

We have implemented the position database as a C++ application. The purpose of the database is to store positions extended by sensor information and allow access by the robot control. Upon request of a position, the extensions of the requested position are evaluated. This includes accessing the sensors connected to the database and retrieving information from them. The interpretation of the sensor data is performed exclusively in the database. The sensor applications only send current values to the database. Lookup in the database is done using the name of the position. If the position does not exist, a null position is sent to the robot and a corresponding flag is set indicating this failure.
The database is encapsulated in a class `PositionManager`. The communication with the robot system is done using the class `RobotConnection`.
We are using four main classes to represent a position in the database: The position itself is realized using the class `ExtendedPosition`. This may be a simple position without any sensor modifications, but an unlimited number of extensions may be attached. Default coordinates are stored in an instance of the class `BasicPosition`. All types of extensions are inherited from the class `Extension` which accesses the class `Sensor` to extract the current sensor data. This data is encapsulated into a generic class `SensorData` to allow universal retrieval and access procedures as well as enable us to integrate additional sensors into the system.
The interrelationship between these classes is illustrated in Figure 7.2. It should be noted that all variables are set to private by default and that there are corresponding *set*- and *get*-operations for each of them. These functions as well as other supporting functions are not included in the diagram to maintain clarity. The function `getPosition` of the class `ExtendedPosition` returns a tuple. The second parameter serves as a flag indicating if a search takes place and that additional positions will follow. The position manager then coordinates the communication with the robot system accordingly. If this flag is not set, only one position is returned.

7.3.1 Sensors and Sensor Data Transformations

We have created programs for every sensor to access the data and sent it to the position database using TCP/IP. Each program runs as a separate process. The syntax to retrieve data is consistent, which allows us to integrate new sensors directly into the system.
In the database application, this information is accessed using the abstract superclass `Sensor` (see Figure 7.2). Three sensor classes inherit from the superclass. The superclass

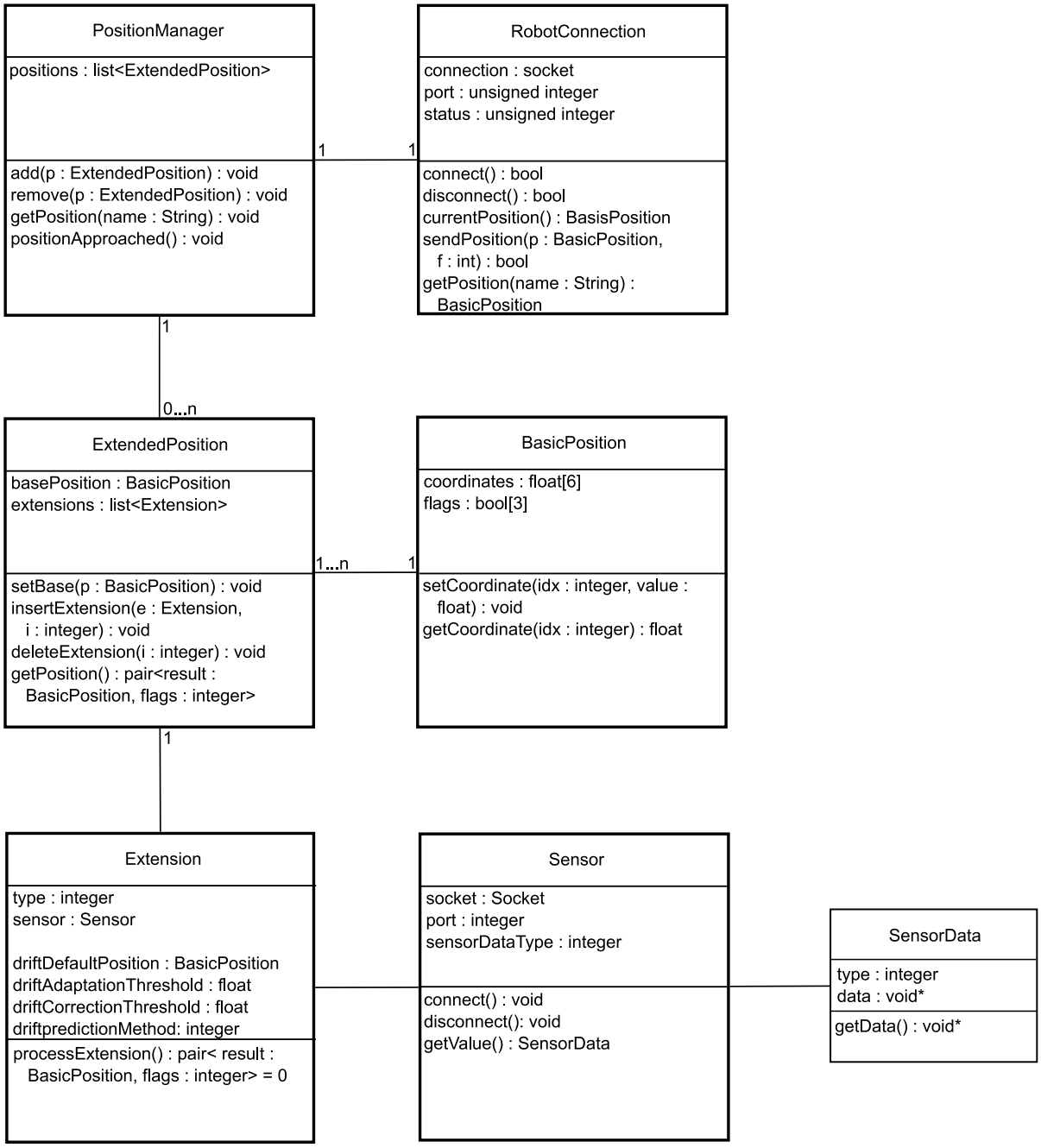


Figure 7.2: Class diagram of the position database.

is abstract, because the procedure to request data and fill the container `SensorData` must be adjusted to the type of sensor.
All sensor data is encapsulated in an instance of the class `SensorData` regardless of its type. This type is stored in the class as well. This allows for an universal algorithm to process the sensor data into either a Cartesian description or a Boolean decision. The type identifier is used to prevent false processing of the data.
All classes that transform sensor data into information are inherited from the abstract super class `SensorTransformation`. These are either `Changefunctions` or `Conditions` (see Figure 7.3).
The class `Condition` is an abstract class as well. Conditions are split into `SensorConditons` to process sensor information and `And-`, `Or-` and `Not-Conditions` that use other conditions to create more complex conditions. The class `SensorCondition` allows the user to set a compare value and to specify if the sensor value must be either larger than, less than or equal to the (absolute) sensor value in order for the condition to evaluate to true. This default implementation can be used for sensor data that is already in the form of numbers of any kind. For more complex sensor data, e.g. images, the developer must derive a new class from this class and overload the evaluate function accordingly.
Geometric change functions are realized in a likewise named class. Since the exact implementation of such functions strongly depends on the task and the type of sensor used, it is abstract as well. The developer must derive it from this class and implement the function `getPosition`. But in many cases some default algorithms can be used to create change functions, especially for sensors providing us with information in the form of numbers. Because of this, we have created pre-set classes for standard change functions.

- The class `LinearTransform` can be parametrized in such a way that we can extract a specific number from the sensor data and apply a linear transformation of the form $f(x) = a + b \cdot x$ of this value to a specific coordinate. Since extensions are processed in sequential order, more complex change functions can be created by concatenating multiple linear transformations.

- The class `FunctionApproximator` realizes a one-dimensional function approximation with a series of pre-defined interpolation algorithms. This class can be used if the connection between the sensor data and resulting position is not linear. We used this class to create a change function for the distance sensors in this work. Again, this class takes one specific value from the sensor data and modifies one coordinate of the resulting position, but more complex change functions for more than one dimension can be created easily by concatenating multiple instances of this class.

- The third class, `SecantMethod`, was created to realize a change function based on the secant method. This class can be used for search motions that are based on this method. The main reason to use this class is to learn more complex change functions as outlined in Section 5.5. This class will provide the necessary corrections for such a search.

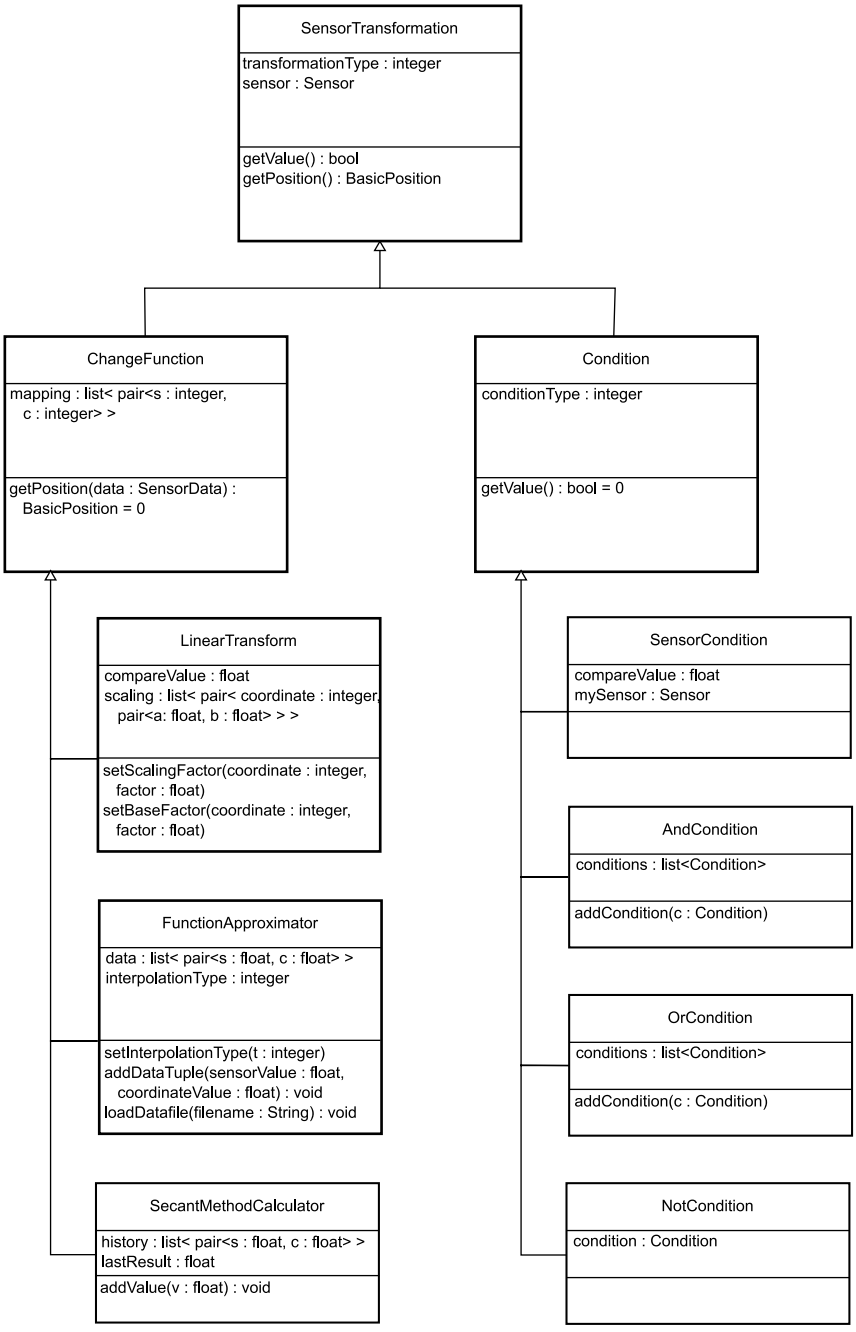


Figure 7.3: Class diagram with inheritances for all modules involved in sensor data processing.

- Another class not shown in the diagram is a class `ImageProcessor`, which was created for change functions that work on images gained from cameras. Here pre-defined change functions are hard to realize, but in this class we have implemented a series of standard algorithms to pre-process image data, such as filtering and Fourier transformation. While this class itself cannot be used directly to create a specific change function, it will help a developer creating a class derived from this class.

In addition to the position manager, there are also databases to maintain all sensors and conditions. But these are only required for the graphical user interface, which will be discussed in Section 7.4. That is why they are not included in any diagram here.

7.3.2 Extensions

Unlike sensor data transformations, which must be implemented or modified by the developer to tailor a given sensor to a specific task, all extensions to a position presented in the previous chapters are fixed and task independent. Each extension realizes a modification of the basic position. Once again, we have created an abstract super class `Extension`. All specific types of extension will inherit from this class. This class already realizes some functionality, mainly maintaining references to the position manager and setting the default position. All drift functionality is included in this class as well. This includes setting the threshold, specifying the drift recognition algorithm etc. These features can be enabled by the developer. If enabled, drift recognition and adaptation will take place automatically. In case of drift correction, a corresponding flag is set when returning the position to the robot control. The developer can employ the commands outlined in Section 7.2 to check if this flag is set and call the drift correction function. The different functionality when processing a specific extension is realized in the function `processExtension` which is just a stub in this class. This function will be called by the position manager when an extension is evaluated.
The class `PositionExtension` realizes the most basic extension possible as described in Section 4.4.2. The developer specifies a change function and the robot will correct the position according to the result of this function.
The class `ClassifierExtension` behaves similarly. Here, the developer adds a series of conditions and corresponding positions. This extension checks each condition and, if it evaluates to true, processes the corresponding position. These positions may be augmented by extensions as well. This enables the developer to create extensions that will only be executed if a certain condition holds. It should be noted that all extensions attached to a position behind a classifier extension will be discarded. This is because a new position will be selected by the classifier which may cause the robot to move to a different position in the workspace. If the developer wishes to employ other extensions in addition to a classifier extension, these must be placed in front of the classifier. This will cause all positions in the classifier to be modified by this extension as well.
Both types of search are derived from another abstract class `SearchExtension` that in turn is derived from `Extension`. The basic parameters for both kinds of search are set in

this abstract class. This includes the terminating condition, the search range and other parameters. The class `BlindSearch` realizes a blind search. Here, the developer must only specify if adaptive path generation shall be enabled or if some default path shall be used in every execution. Apart from that no other parameters must be provided. The class `InformedSearch` is used to create informed searches. Since the calculation of the next position is done in a change function, the developer need not derive new classes from this class, but only set some parameters and the change function. The search algorithm itself does not need to be modified.
If searches are used to learn change functions, a search extension must be connected to a change function in order to inform that change function about the correct position of the sensor value. With the given implementation this is only sensible if the change function is an instance of `FunctionApproximator`, since this class calculates the change function using a set of data tuples. To realize this functionality, the developer can set corresponding flags using `link` in the abstract class `SearchExtension`.

7.3.3 Processing an Extension

We have described the algorithm to process an extension in Sections 4.4.2 and 5.1.3. In this section, we will only describe a minor modification that becomes necessary because we use a distributed system.
To prevent the various components from busy waiting for responses from sensors or the database, we have used a state machine instead of direct function calls. This means that all requests (either by the robot to the database or by the database to a sensor) are send as a command to the computer running that specific component. The caller returns to an idle state and waits for a response from the callee. Timeout mechanisms were used where appropriate. This functionality is realized using the QT library [8] that provides a *signal/slot* mechanism to easily create state machines.

7.4 A Graphical User Interface to Manage the Database

Up to this point, creating and adding positions to the database must be done by textual programming and re-compiling the database application.
To further alleviate development especially for non-computer scientists, we have added a *graphical user interface* (GUI) to the database application. With this GUI, a user can create, modify and delete positions in the database without the need for textual programming. When the application is started, the database connects to all sensors and to the robot. In a main dialog the user sees all positions in the database (see Figure 7.5). The user can select a position and is shown the default values and all extensions attached to that position. These can be ordered by a series of buttons. The user can add, modiffy and order positions and extensions by pressing the appropriate buttons.
To create a new extension, the user is presented with a series of dialogs guiding him through

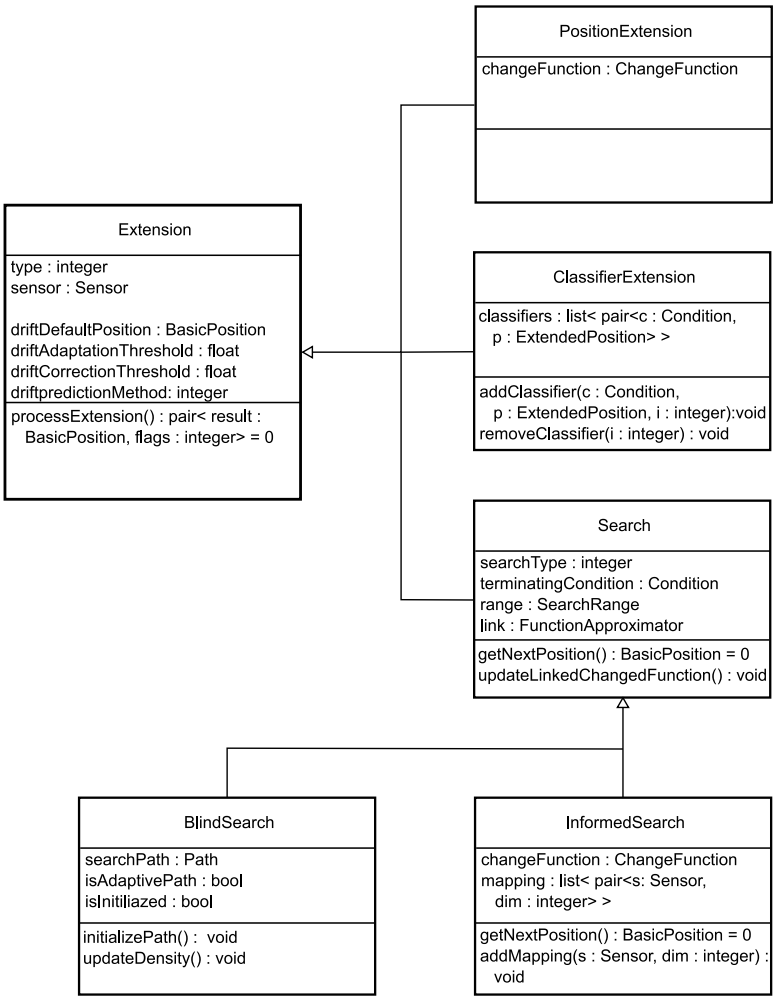


Figure 7.4: Inheritance graph of all types of extensions. The abstract super class *Extension* is used to link an extension to a position in the database. The actual functionality of the different extensions is encoded in the derived classes.

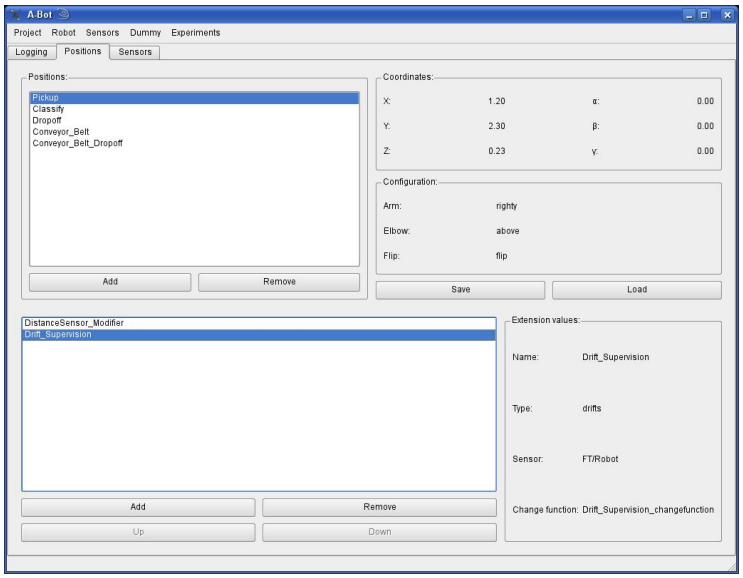


Figure 7.5: The main screen of the graphical user interface. All positions currently stored in the database are shown in the top half of the window. The extensions belonging to the currently selected position are shown in the bottom half of the window. The user can add and modify and order positions and extensions by a series of buttons.

the creation process (see Figure 7.6). In a first step, the user selects the type of extension and depending on this selection, the extension is parametrized in the following dialog screens. This is possible because the functionality of all extensions is fixed. Change functions are created in this dialog as well. The user selects the type of function and enters the desired parameters. A separate dialog is used to create a condition, similar to the extension dialog.
With this system, the user can manage the database without having to resort to textual programming. The drawback of this approach is that it is not possible to create new change functions or conditions by deriving them from the base classes. To overcome this problem, a dynamic linking of special libraries during runtime may be used. With this solution, the user must program the new functionality and compile it into a library. This feature can also be used to allow manufacturers of new sensors to create such libraries and ship them with the sensor. The user can then simply add these libraries to the system when physically installing the sensor in the workspace. In this case the user does not even have to create change functions and special conditions on his own. These can then be loaded by the database application to enable the user to select these new functions in the GUI. We have not added this functionality to the system, since it will only improve usability but not the general features of the system.

7.5 Experimental Evaluation

We have tested our concept using this software. In order to do so, we have created a series of example tasks, each of them dealing with a different aspect of flexible and adaptive robot programming.

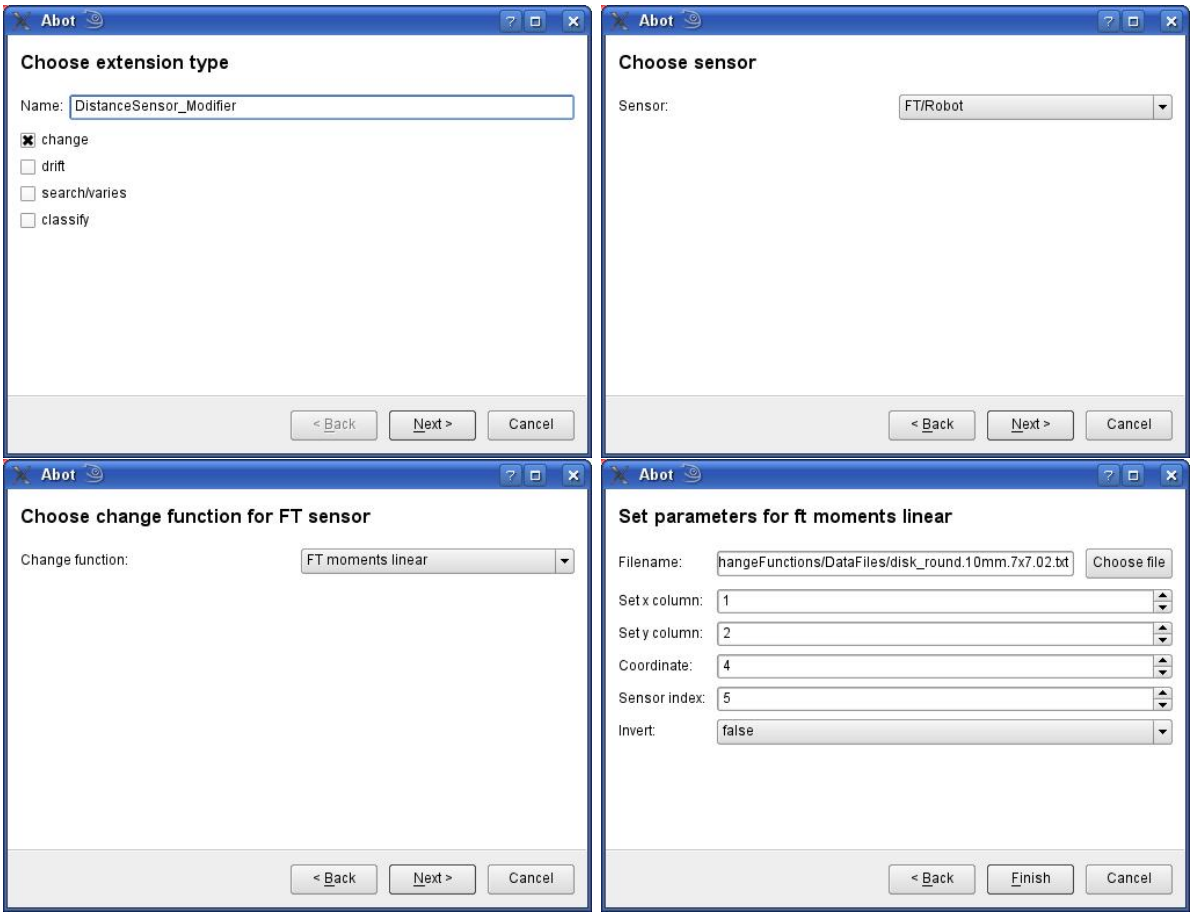


Figure 7.6: Dialog wizard to create new extensions. Top left: On a first screen, the user must select the type of extension and name it. Top right: On the next screen, the sensor belonging to that extension is chosen. Bottom left: Next, a suitable type of change function is selected. Bottom right: The selected change function is parametrized.

7.5.1 Test Cases

We have created seven different tasks, each of them to solve a different task. To solve them, two of the three sensors must be used: The force/torque sensor and the distance sensor. We have not created tasks that specially need a camera. Programming change functions for these would have exceeded the scope of these experiments. To get an idea about the complexity of creating such a change function we had a student of computer science create a classifier to detect different object shapes within a camera image. This task took the student approximately 50 hours and resulted in a program of roughly 6000 lines of code. This is good indicator that a non-expert developer would need at least professional software libraries to allow creation of such functions within a realistic time frame. We have decided to leave the camera out because of the complexity to work with such libraries and focus on the other sensors instead.
Each task shall be solved twice: Once using the robot's built in programming language V+; and once more using the application described in the previous chapters. Solutions with our application were solved using textual programming as well and not the GUI. This is because we were interested to see how the probands would create the code describing the positions in the database. Another issue was that typing a solution in one language and clicking it together in the other devalues the measured time that is taken for each solution. Due to these two aspects, we have asked the probands to create the solutions using our framework textually as well. It is important to note that the workspace will not be modified in any of the solutions to the tasks. Neither will there will be any re-arrangement of objects nor the introduction of mechanical devices. All tasks are solved using existing sensors only. Note, that all of these tasks can be solved by evaluating the sensors when the position is requested by the robot. In no case is it necessary to toggle a pre-processing of the position because the robot occludes an object from a sensor. While we cannot prove that this will be the case for every possible handling task, there is strong evidence, that the concept of 'just in time' sensor evaluation, as outlined in Section 4.4.5, is sufficient and explicit pre-processing of positions can be avoided by careful positioning of the sensors in the workspace and choosing adequate positions.

E1 - Pick

The goal of this task is to pick up an object which is delivered to the robot on a conveyor belt (see Figure 7.7, left). The belt stops when the object passes a light barrier installed at the end of the belt. This means that the y-coordinate of the pickup position is fixed. But a variation in the x-coordinate of the pickup position is allowed, implying that the object may lie anywhere between the light barrier.
To detect the object the distance sensor shall be used. The robot shall be able to grasp the object safely by evaluating the sensor signal and applying the information to the default position. The desired accuracy is 1 cm as this is also the general accuracy of the sensor. The data curve of the sensor shows that the relation between distance and sensor signal is not linear (see Figure 3.4, right). So a more complex change function must be created.

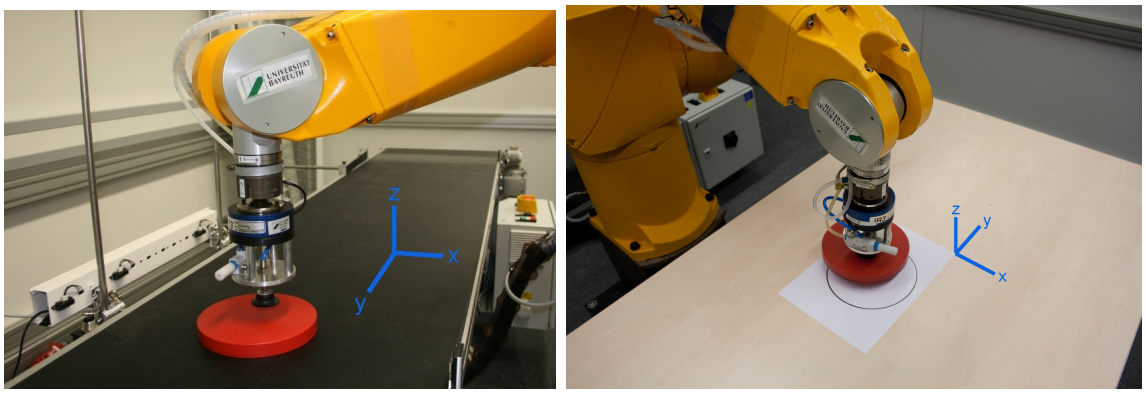


Figure 7.7: Left: Setup for experiments E1 and E2. The position and drift of the disk is measured using distance sensors mounted to the side of the conveyor belt. Right: Setup for experiment E3. The grasped disk shall be placed accurately in the center of the shape below.

Key aspects of this task are to realize that the data curve of the sensor is not linear and to create a reasonable change function. In addition, the pickup position must be modified by adding a `PositionExtension` and parametrizing this extension accordingly. The sensor is used preparatoryly in this task.

E2 - Drift Recognition

The goal of this task is to implement a drift recognition. We set up the conveyor belt as illustrated in the experiments of Section 6.1. Here the task is again to locate and grasp the object on the belt. But this time, the robot shall place the object on the other end of the conveyor belt. The conveyor belt was set up in such a way that a drift occurs. This drift shall be recognized across multiple executions and a corresponding correction function shall be created. Because the distance sensor is used in a preparatory way, an adaptation by the drift prediction is not necessary.
Key aspects of this task are to employ persistent storage of the pickup position across multiple executions by using a history. The history must be used to check for a drift and toggle a corrective motion. This correction can be executed at various points in the program: Either when the object is grasped or when it is placed on the belt at the other end. We did not specify which of the two options shall be preferred. When realizing this task, the drift functionality of the class `Extension` must be toggled and parametrized. The sensor is used preparatoryly in this task.

E3 - Place

The goal of this task is to pick up a round disk and place it as accurately as possible on a table. The disk is taken off the conveyor belt. Given the accuracy of the distance sensor and the delay when stopping the belt triggered by the light barrier, the actual grasp position may vary by 1 cm from the disk's center along the x- and y-dimension. The desired accuracy

of the placement shall be 1 mm in both dimensions (see Figure 7.7, right). We want to use the force/torque sensor to measure the offsets when the disk is held and calculate the necessary offset to place the disk accurately.
It should be noted that this problem cannot be overcome by simply using a two fingered gripper instead of a vacuum gripper. While this would compensate the disk's variation in the x-coordinate, there would still be a variation in the y-coordinate. In addition, this solution would constitute a mechanical modification of the workspace which we have ruled out.
Key aspects of this task are to identity the fact that the moments along the x- and y-axis of the force/torque sensor can be used to compute the variation. There are a series of important features that must be identified to solve this task:

- In order to get an exact measurement, the disk must be held freely and perpendicular to the ground so the measurements are correct.
- The measured moment along the x-axis describes the variation along the y-axis and vice versa.
- The measured moments are linearly independent from each other.
- The change function to convert a moment into a Cartesian description is linear.

To solve this task, we can either use two instances of `PositionExtension`, each of them calculating the variation in one dimension using the existing class `LinearTransformation`, or we must derive a new class from the `ChangeFunction` calculating both variations in one pass. In this case, one extension is sufficient. The sensor is used concurrently in this task.

E4 - Classifying Objects

The goal of this task is to pick up an object from the conveyor belt, determine its shape and then place it in a corresponding container on the table. There are four different shapes: A circle, a square, a parallelogram and a pentagram (see Figure 7.8). For this task, we do not care about possible variations in the offset of the disk's center or the disk's rotation.
Typically, this task would be solved by using a camera. But, as outlined above, solving this by creating a change function to analyze an image for the shape would be too complex for this series of experiments. Because of this, a different approach is used here. The disks are all made of the same material but each disk differs in weight. So the disks can be classified using the force/torque sensor as well.
Key aspects of this task are to use the `ClassifierExtension` and create four different conditions to determine if the held disk is of a specific shape. These conditions must then be mapped to the four possible boxes. The sensor is used concurrently in this task.

E5 - One-Dimensional Search

The goal of this task is to place an object on another object on the table. The position of the object on the table is known, but not its height (see Figure 7.8). Because of this a

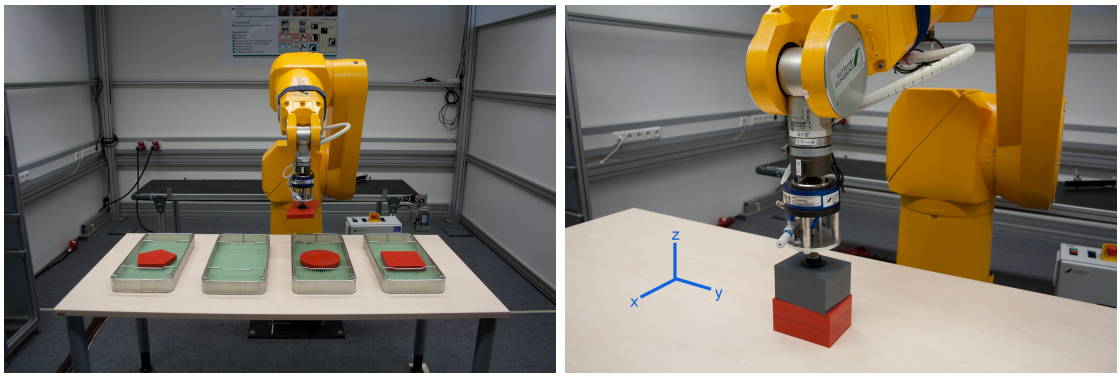


Figure 7.8: Left: Setup for experiment E4. The disks shall be placed in the appropriate boxes according to their weight. Right: Setup for experiment E5. The robot shall perform a guarded move to place the grey block on top of the red block.

guarded move must be performed where the robot stops when the held object touches the other object.
Key aspects of this task are to realize that a blind search must be performed to solve this task using the force/torque sensor to stop the search when a significant contact force is measured. The search space is one-dimensional and lies along the z-axis of the table's coordinate system. A suitable search range must be defined and a terminating condition must be created. The sensor is used concurrently in this task.
It should be noted that normally this task would be solved using the guarded move functionality of the robot programming language. Here, the robot would evaluate the force/torque sensor automatically when performing the motion and trigger a stop signal. This realization will cause the robot to move more smoothly and achieve its goal much faster than with our system. But an implementation with the Adept extension to the V+ robot language takes about 14 lines of code. The parametrization of these is by no means trivial. Unskilled developers will solve this task faster by creating a simple step-by-step search which evaluates the terminating condition by itself. The disadvantage of a longer execution time will be compensated by the faster development time if the number of repetitions is sufficiently small.

E6 - Two-Dimensional Search

The goal of this task is to locate a hole in a plate on the table in order to start an insertion process for a peg. No cameras are available, so the localization with a `PositionExtension` is impossible. Instead the force/torques sensor must be employed by conducting a blind search across the plate (see Figure 7.9). The actual insertion of the peg will be done in the next experiment.
Key aspects of this task are to create a two-dimensional search with an efficient search path in terms of development time. We do not care if the path is generated adaptively or if a standard path is used. The terminating condition is of less impact here (but must be

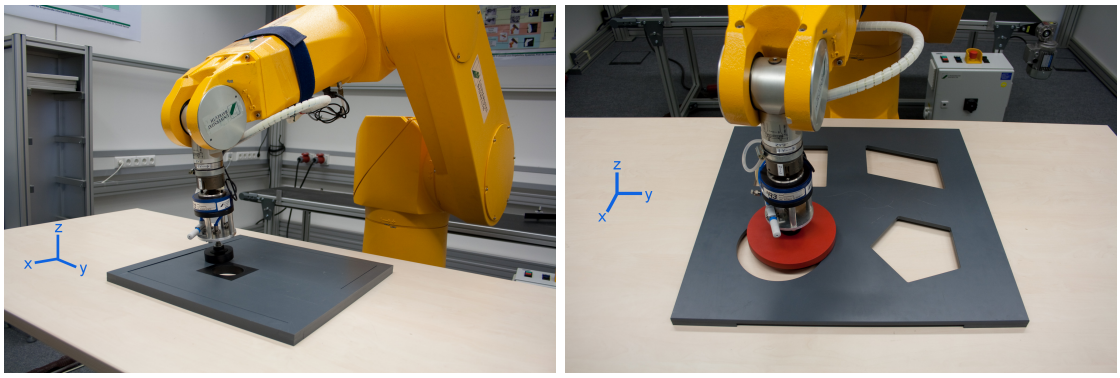


Figure 7.9: Left: Setup for experiment E6. The robot must locate the hole in the plate using a two-dimensional blind search. Right Setup for experiment E7. The robot shall insert the round disk into the corresponding hole.

created nonetheless). The sensor is used concurrently in this task.
It should be noted that unlike in experiment E5, a realization of the search with guarded moves quickly becomes very complex.

E7 - Insertion

The goal of this task is to insert a peg into a hole using a force/torque sensor. Firstly, we use the small peg shown in Figure 7.9 on the left but come to realize that it is too small to allow for a sufficiently easy solution for our probands. A feasible solution would involve a tilting strategy and a four-dimensional change function, processing not only moments but also forces. The problem gets significantly easier if a larger peg is used (see Figure 7.9, right). In this case a solution can be found that only needs to evaluate the moments of the sensor. As the focus of the experiments is to measure the improvement compared to classical programming and not to create algorithms for efficient insertion, we have opted for the bigger peg.[1]
Key aspects of this task are to set up an informed search in a two-dimensional space using the class `InformedSearch`. In addition to the terminating condition a change function must be created to compute the next position in the search. Because of the size of the peg and the type of gripper two linear correction change function may be used for a very simple insertion strategy. But other approaches were allowed as well.

[1]But this shows one of the big problems with informed searches. These quickly become very difficult. So the difficult part is to come up with a clever solution for the change function to compute the next position. At this point insertion skill libraries may become an interesting extension. This aspect will be discussed in Chapter 8.3.2.

7.5.2 Probands

We have five students as probands for the experiments. Each student solved every task twice; with the built-in programming language V+ and with our implementation of the new framework. The order of the experiments changed randomly for every student. Exceptions were E1 and E2, which were always solved in that order, and E5 was always solved before E6 and E7.
Two students were bachelor students in computer science, two were master students in computer science and one student was a master student in physics. The computer science students aiming for the master's degree had heard lectures about sensor data processing and robotics. The other three students only had general knowledge in programming using object oriented languages like C++. None of the students had ever worked with a robot before.
In a first step all probands were taught to program the robot using V+. This included moving the robot using the teach panel and the command line. All students learned how to create static robot programs, including if-, for- and while-constructs. This also included setting the robot's speed during execution of a task. With this knowledge all students were capable of working with the robot in a basic way.
There was a supervisor present during the whole time the probands worked on a task. Apart from ensuring that the robot was used safely, the supervisor answered questions regarding the programming syntax and structure of both languages, V+ and our implementation. There were no hints in terms of algorithmic solutions to any of the tasks. But data files with sensor calibration data were provided when the proband had explained correctly in which way he would create these files. This mainly concerns the distance sensor, as the change function is not linear for this sensor.
For each task, the probands were given a static robot program that solved the task correctly, provided the objects were not subject to any kind of variation or drift. This included the basic positions. The proband then had to modify the program to incorporate sensors and modify the given positions accordingly.
We have recorded the time it took the proband to create a correct solution to the task and the lines of code of the solution. When measuring the lines of code, we only counted new lines added to either the program or the position database not including comments, empty lines and lines containing only brackets. After the probands had created solutions to all tasks, they had to fill out a questionnaire asking about their experience with the new framework.

7.5.3 Results

The results of the experiments are depicted in Tables 7.1 and 7.2. Table 7.1 shows the average time required to create a solution for a specific task in minutes (columns two and four). In addition, we have given the standard deviation s (columns three and five). The gain is computed as the percentage of development time saved when using our framework instead of pure textual programming (column five).

Experiment	Textual		A-Bot		
	m	s	m	s	Gain
E1 - Pick	187.0	75.9	70.6	10.8	62%
E2 - Pick with drift recognition	31.7	24.6	10.0	6.5	68%
E3 - Place	63.8	43.8	50.5	33.3	21%
E4 - Classify object by weight	52.8	34.4	20.5	11.3	61%
E5 - 1-dimensional blind search	18.2	4.3	9.4	4.4	48%
E6 - 2-dimensional blind search	84.2	42.0	16.4	13.8	81%
E7 - Insertion of round disk	105.4	79.7	25.0	19.1	76%
Average of all experiments	77.6	43.5	28.9	14.2	63%

Table 7.1: Mean time m and standard deviation s in minutes required to solve the experiments

Experiment	Textual		A-Bot		
	m	s	m	s	Gain
E1 - Pick	185.2	7.6	40.0	2.1	78%
E2 - Pick with drift recognition	23.7	13.6	13.0	7.2	45%
E3 - Place	20.3	9.1	10.5	4.8	48%
E4 - Classify object by weight	25.0	11.8	12.0	5.9	53%
E5 - 1-dimensional blind search	19.4	2.1	5.2	1.1	73%
E6 - 2-dimensional blind search	54.0	8.0	10.8	3.9	80%
E7 - Insertion of round disk	27.0	4.7	7.4	1.3	73%
Average of all experiments	50.7	8.1	14.1	3.8	72%

Table 7.2: Mean m and standard deviation s lines of code required to solve the experiments

Table 7.2 shows the average number of lines of code for a solution for a specific task (columns two and four). Again, we have also given the standard deviation s (columns three and five). The gain is computed as the percentage of lines of code required less when using our framework instead of pure textual programming. (column five). For experiment 2, we have only counted the lines of code related to drift recognition and correction. Graphical plots of these results are shown in Figure 7.10.

We can see that the new framework allows for significant savings both in time required and lines of code. The new framework with pre-set extensions simplifies the creation of solutions in experiments where either elaborate change functions (E1) or complex mechanisms had to be realized (E2, E6, E7). In experiments with simpler operations (E3, E5) this is of less impact. This is because here the algorithm itself is straightforward with little room for variations in terms of programming.

The long time required and high number of lines of code for experiment E1 is caused by the complex change function of the distance sensor. Here the probands had to create a lookup table to convert the sensor value into a Cartesian description. This takes time and

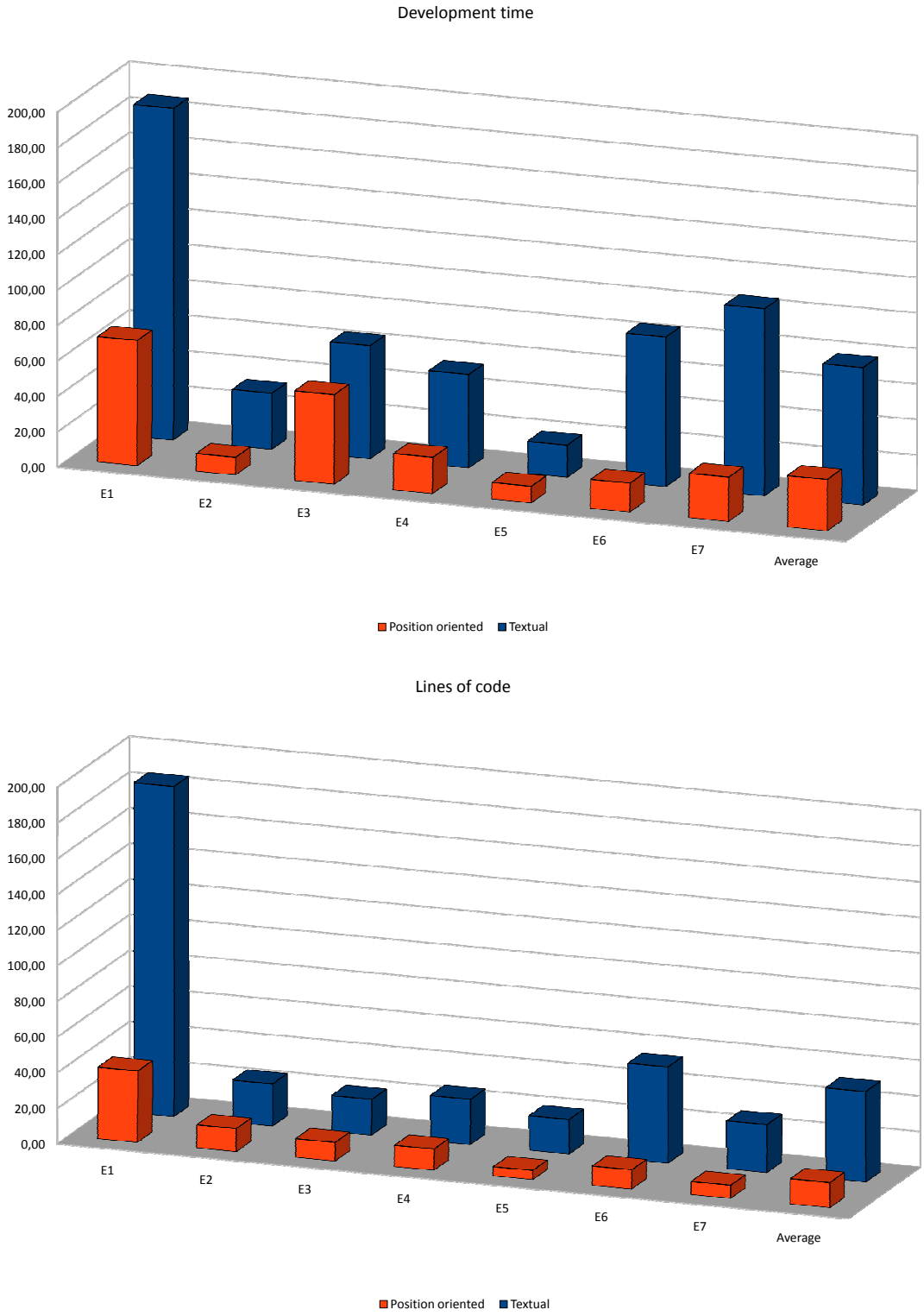


Figure 7.10: Results of the experiments conducted. Top: Average time required by the probands to create a valid solution. Bottom: Average number of lines of code of the solutions.

needs a lot of lines of code.
The biggest gain could be achieved for the two-dimensional search in experiment E6. This is because the algorithm for path generation requires the developer to check for a series of exceptions. This results in longer programming time and more lines of code. As the path generation for blind searches is completely encapsulated in our framework, the developer saves a lot of time and lines of code.
In summary, the results of the measurements are promising, as we can see a significant decrease in both measures with an average of more than 50 percent. But these results must be interpreted carefully. For one, all students solved the task twice. This means that experiences made in the first attempt (textual programming in V+) could be re-used in the solution with the new framework. Another fact is that a lot of time was required to measure the data curve of the distance sensor and learn to interpret its values. But this is done only once and does not need to be repeated in the second implementation. So the measured time to solve a task should only be taken as an indicator.
To get more realistic values, more thorough experiments must be made. This includes using probands and tasks from industrial settings as well as control groups. In addition, every proband should solve a task with only one style of programming. But this requires a much larger number of probands.

Questionnaire

We asked all probands about their impressions on the new framework. The questions were split into two categories.
In the first category we asked the probands to self-assess their programming knowledge, the practicality of the tasks given and their rating of the A-Bot system. This included general programming, robot programming, sensor data processing and practicality of the tasks. The probands could give answers in a range from 0 to 10. A zero indicated he had no knowledge at all or the task is totally unrealistic, while a ten indicated that the student considered himself to be an expert or that the task will be the same in actual industrial environments. The averaged results as well as the standard deviation are given in Table 7.3.
All probands considered themselves to be experienced in general programming, but only had limited knowledge in robot programming. The master students thought themselves to be reasonably skilled in sensor data processing, but the other students were a lot more sceptical. All tasks were considered to be fairly easy yet realistic in a general industrial usability. In general, the students thought the idea of position oriented programming to be a sensible approach allowing easier and faster programming of sensor based robots. In a final rating the students considered the new framework to be helpful and prefered it to textual programming.
In the second category we asked the probands to compare the new framework to purely textual programming. Here, the probands could give answers in the range from -3 to 3, where a -3 indicated the new framework to be a lot worse than textual programming and 3 to be a lot better. If the two approaches compared similarly a value of 0 should be given.

	Experience			Experiments		Practical use		
	General	Robot programming	Sensor data processing	Difficulty	Realism	Position orientation	Ease of programming	Preference of A-Bot
m	7.0	3.9	4.8	4.5	7.3	8.8	7.5	8.5
s	0.0	1.3	2.2	1.0	1.7	1.0	0.6	1.0

Table 7.3: Mean (m) and standard deviation (s) of the probands self assessment regrading their knowledge in robot programming, the realism of the tasks and their rating of the A-Bot system. The values may be within a range from 0 to 10.

The averaged results as well as the standard deviation are given in Table 7.4.
To sum up the results: All probands considered the new framework to be an improvement to textual programming in all categories. Especially, the development speed and the comfort were rated high. The views on reusability and amount of code required differed, but were positive nonetheless. The solutions were considered to have been intuitive and clearly laid out. Some probands considered the understandability to be difficult at some points. When asked for details, they explained that some of the terms used were ambiguous to them.

7.6 Conclusions

In this chapter we have presented an implementation of the concept developed in the previous chapters. We argued that, due to hardware limitations of the robot controller, the implementation must be realized in form of a distributed system with all sensors and the database on separate computers. This allows for a highly modular system and new sensors can be added dynamically. A new robot command `vmove` is introduced to access the database. Its syntax is similar to existing move commands.
We have outlined that the basic functionality of the database is contained in abstract classes that can be used by the developer to create new change functions and conditions tailored to the task. The general algorithm to process the different types of variations is encapsulated into extension classes that only need to be parametrized. Because of this, the developer does not have to create new algorithms to compensate variations.
We have added a graphical user interface to manage the database and add, modify and delete positions. This is done with a series of dialogs enabling the developer to create positions with multiple extensions easily. The downside is, that with this user interface no new change functions and conditions can be created. This must still be achieved by textual

	Intuitivity	Comfort	Reusability	Clarity of style	Understandability	Development speed	Amount of code required
m	1.8	2.3	1.5	1.5	1.0	2.0	1.5
s	0.5	0.5	1.3	0.6	0.8	0.8	1.0

Table 7.4: Mean (m) and standard deviation (s) of the probands rating of the A-Bot system compared to pure textual programming with a range of values from -3 to 3.

programming.
To evaluate our system, we have created a series of example tasks focusing on different aspects of flexible and adaptive robot programming. We have tested our framework and compared it to pure textual programming by having five probands solving these tasks. We show that with our framework these tasks can be solved faster and with fewer lines of code. In addition, we have conducted interviews with the probands and found the acceptance of the new framework very high.

Chapter 8

Conclusion

In this chapter, we give a complete overview of the work presented here and consider its limitations as well as possible additions. In Section 8.1, we summarize our work. In Section 8.2, we discuss the possible limitations of the approach presented here. In Section 8.3, we outline possible additions to our work to further increase the usability of the presented concept.

8.1 Summary

The purpose of this work was to find a new approach to intuitive sensor based robot programming for non-experts. To alleviate the apllication of robots in industrial applications we have researched this area and created a new way of programming for non-experts requiring only minimal knowledge in robotics. First considering the downsides of already existing systems we saw that they are either geared towards experts in both robot programming and sensor data processing or tailored to specific task domains or types of sensors.
In this work, we focussed on industrial handling tasks, that place a large emphasis on positions and orientations whereas trajectories are of minor importance. We have assumed that a basic robot program is given that is capable of solving the task in a static workspace. Our goal was to find intuitive ways to integrate external sensors into the program to enable the robot to react flexibly and adaptively to changes in the workspace.
In a first step, we have shown that sensor information can be classified into two categories for our purpose: Geometric properties and conditional properties. Geometric properties describe some features of a workspace change that can be expressed in Cartesian coordinates. Conditional properties are used to determine features of objects that are either present or not and can thus be expressed as a binary value. We have introduced the concept of sensor transformation functions to transform sensor information into one of these two abstract descriptions. This allows us to employ universal approaches for flexibility and adaptivity algorithms.
Based on these abstract transformations we have created the concept of position oriented robot programming. This concept differs from existing approaches in the sense that modi-

fications of the robot's behaviour based on sensor information are not integrated into the robot program but added to specific positions using the concept of extensions. When a position is approached within the program, the position database evaluates all extensions attached to the position and modifies it accordingly. This approach yields the advantage that there is a strict separation between sensor data processing and robot commands. This allows for increased maintainability as well as the re-use and modification of the program at later points.
There are different ways in which an extension can modify a position. These are closely related to the type of sensor transformation available for that sensor and if the robot can compensate the workspace alteration directly or needs to perform multiple corrections. We have presented four types of extensions for these different cases. Change and classifier extensions are used for direct compensations based on geometric sensor transformations and conditional properties respectively. Blind and informed searches are used to compensate alterations that require multiple corrections by the robot. The advantage of these extensions is, that the correction algorithm is fixed and does not need to be modified by the developer. The developer only needs to set suitable sensor transformations and other parameters to fit the extension to the task.
In addition to the flexibility that can be integrated into the robot program, we have also explored approaches for adaptive behaviour of the robot. This included implicit learning of geometric sensor transformations, the automated creation of adaptive search paths based on probability distributions and online drift adaptation and correction. All of these features are integrated into the framework and can be enabled by the developer without the need to design intricate algorithms. The purpose of these adaptive features is to further decrease the time needed for development and execution of the program as well as to allow the robot to compensate minor errors when executing the task.
We have shown the feasibility of our concept by creating a C++ application realizing the robot position database. We have extended the robot programming language V+ to access this database. We have created a series of example tasks representing typical handling problems in industrial contexts and had five probands solve these twice. One using only the standard version of the V+ language; the second time using the new position database and our extension to V+. We showed that the new concept allows for significant savings in development time and lines of code required to solve the task. We saw that the probands prefer the new system to the standard programming language in terms of usability, intuitivity and comfort, by using a questionnaire.
In summary, we can say that the approach presented here allows a developer who is unskilled in sensor data processing to extend static robot programs with external sensors easily and in an intuitive way. The approach is geared towards industrial handling tasks and focusses on the manipulations of positions and orientations of objects. Trajectories are explicitly not covered with this approach. Regarding the type sensors that can be used with this approach, we can say that there are no limitations provided the sensor is capable of measuring an alteration in the workspace and expressing this in terms of either a Cartesian description or a Boolean condition.

8.2 Discussion

In this section we discuss some aspects of the approach presented in the previous chapters. This includes evaluating the benefits of the proposed concept and the required knowledge in robot programming and sensor data processing by the developer. We will also discuss the need for new types of extension that are not covered in this work and the general applicability of the presented concept.

8.2.1 Types of Extensions

The four different extensions presented in this work cover the complete range of applications for handling tasks in this area. There is no need for more types of extensions. For clarification we can demonstrate this along the following argumentation:
In case of a direct compensation, a change extension (Section 4.4.3) is used if the sensor information can be transformed into a Cartesian description of the alteration. If the information is present in terms of conditional properties, a classifier extension (Section 4.4.4) is used instead. The final option for direct compensation is to employ both types of information, a Cartesian description and a conditional property. There is no need for an extension that requires both types of information for direct compensation, because multiple extensions can be attached to a position, which are all processed before the position is approached. So this type of extension can be simulated using multiple change and classifier extensions.
In case of iterative compensation, we must employ some kind of search motion. A conditional property is mandatory, otherwise we cannot determine the end of the compensation. If there is only a geometric sensor transformation function available, we cannot employ a search because we would not know when to terminate. The only option here is to create a conditional performing a subtraction of the result of this function from a target position and check if the result is below a given threshold. But then this is a conditional property. If we only use this conditional property, a blind search (Section 5.3) is sufficient. If we also use the geometric sensor transformation as well, an informed search (Section 5.4) takes place. So there can be no extension for iterative compensation that only uses a geometric sensor transformation.

8.2.2 Non-Sequential Processing of Extensions

In Section 4.4.2 we have explained that extensions attached to a position are evaluated in sequential order when that position is accessed in the database. At this point we discuss if there are alternatives to this approach.
One idea is to allow if- and while-constructs when adding extensions to a position. The developer could instruct the database to evaluate an extension only if a condition holds or to repeat the evaluation for a couple of times. But no extra functionality is gained with this upgrade.
A repetition of an extension is already realized within a search extension. There is no need

to execute a search multiple times, since either the goal position will be found during the first execution of the search or the search will fail completely. It does not make sense to repeatedly evaluate a change or a classifier extension. These two extensions are used for direct compensation. This implies, that the sensor information can be applied directly to the default position. Since the robot does not move during evaluation, the sensor signal will not change. Because of that the result of the evaluation will always be the same[1].
An if-construct would represent a conditional execution of an extension. This can be simulated using a classifier extension and a virtual positions. The condition to evaluate an extension is linked to a virtual position. The extension itself is then attached to this position, but not to the original position. So if this condition holds, the virtual position is selected and the extension is evaluated. The difference between a classifier extension and are real if-construct is that the extension allows us to maintain the strict separation between the robot program and the sensor data processing. If we used if- and while-constructs, we would create a second programming language in the position database to evaluate the extensions in a specific order.
Because while-loops are unnecessary and if-constructs can be realized using classifier extensions, the sequential processing of extensions attached to a position is sufficient.

8.2.3 Required Knowledge of the Developer

To use this framework, the developer must possess basic robot programming skills, that is he should know the basic commands to move the robot, set its speed and so on. The main part of this is that he knows how positions are described in relation to task frames and how a robot will move to a defined position.
Based on this foundation, the developer must have understood the basic principle of position oriented programming as outlined in Chapter 4. He must know what types of extensions there are and how they are processed. Since the algorithmic processing of all extensions is fixed, there is no need to learn new commands or modify these extensions.
The difficult part is to parametrize these extensions. In a first step, the developer must decide which sensor shall be used to compensate the variation and where it is to be placed in the workspace. This depends completely on the task and cannot be alleviated by our system. In the next step, suitable sensor transformation functions must be created. Once again, this is highly dependent on the task. We have outlined general approaches to create these transformation functions in Sections 3.4, 3.5 and 5.4. With these approaches, the developer can create 'simple' transformation functions. The biggest problem is creating geometric sensor transformations for informed searches. Here, the function serves to determine the next position in the search. The underlying search algorithm itself is simple. For complex operations, e.g. insertion of irregular shaped objects with low tolerances, the 'magic' lies within this function. The definition of an insertion strategy is a complex task requiring lots of experience.

[1] If the object that is supervised by the robot is moving, we are working in a dynamic workspace. The applicability of our framework in such environments is discussed in Section 4.5.

It would be helpful if sensor manufacturers provided libraries containing at least basic transformation functions for their sensor. In addition, there are professional software libraries for special classes of sensors, such as cameras. These will help the developer to a large extent, but cannot free him completely from this task.

8.2.4 Benefits of Position Oriented Programming

The advantages of the concept presented in this work are as follows:

- The concept of position oriented programming is neither geared towards specific types of tasks nor specific types of sensors. In principle, all industrial handling tasks can be solved with this approach using any type of external sensor. This concept can be used for any industrial robot and is not limited by physical capabilities of the robot e.g. the number of joints.
- This concept realizes an expansion for a textual robot programming language. Unlike other approaches, e.g. the task frame formalism (see Section 2.3) a given static robot program can be extended easily by adding extensions to the positions in the database. There is no need to either modify the program or transfer the program to a new programming language.
- The strict separation of robot commands and sensor data processing ensures improved readability and maintainability of the robot program. Sensors may be replaced by other types of sensor without the need to modify the program. The only thing to be done is to set the parameters of the extensions accordingly.
- Adaptivity algorithms are integrated into the system. This allows for an internal optimization in terms of execution time and stability without the need for elaborate modifications by the developer.
- Insofar a sensor is provided with a library containing sensor transformation functions, the developer must only set parameters for these functions but does not need to deal with the development of these functions.

8.2.5 Limits of Position Oriented Programming

This work only deals with the interpretation of sensor information in order to modify positions during the execution of a robot program. Specifically not covered are the problems of choosing the right sensor and placing it in the workspace.
While the framework developed in this work allows the user to add flexibility and adaptivity measures to a static robot program, one problem remains that cannot be alleviated: With this generic approach, the developer must create sensor transformation functions by himself. Because the framework was designed to remain abstract from the specific type of handling task and sensor used, we described some general approaches to create this sensor

transformations. Unfortunately, these functions quickly can become complex, especially for informed searches. Here, the developer must have a precise understanding of how the sensor information shall be used within a sensor transformation function.
This also implies that the developer must use some kind of programming language to create these functions. The graphic user interface created in Section 7.4 is only sufficient if known methods are used for creation of sensor transformation functions. This problem may be overcome at some point by using rule based systems when sensor transformation functions must be created from scratch. Here, the developer may be able to create such functions by designing sets of rules which compute the transformation. We have not pursued this idea, as it exceeds the scope of this work.
As shown in Section 4.5, the developed framework can be used to solve all types of industrial handling tasks, provided that the alteration of the workspace happens at such a speed that the robot has enough time to observe, compute and react to the alteration. But this requirement holds for all robot programs that are modified by external sensors during execution. Successful execution cannot be guaranteed if the analysis of the sensor signal takes too long.
Tasks that require the modification of trajectories, like painting and spot welding, can be solved by our framework only when workarounds are used. Since trajectories are computed within the robot program and altered during runtime based on external sensor information, our approach cannot be used directly. A workaround is to define the trajectory as a position with an attached search extension. The default position is the starting position of the trajectory. The search extension computes the next intermediate point along the trajectory and the terminating condition evaluates to true, when the end of the trajectory is reached. An example of this workaround was given in experiment E5 in Section 7.5. In general, we can say that robot tasks which require the alteration of trajectories can be solved better using other approaches such as the task frame formalism (see Section 2.3).

8.3 Outlook

In this work we have outlined a first approach to extend a given static robot program with external sensors. The presented methods are by no means exhaustive. Additional concepts and expansions can be added to this framework at multiple points. To summarize this work, we outline possible approaches to extend the applicability of this framework for intuitive robot programming with external sensors.

8.3.1 Minor Modifications and Additions to the Framework

Here we discuss some minor additions that can be made to this framework. These are geared towards specific problems occurring in certain cases and may help to further decrease development time. The only reason they were not incorporated into our framework is that no new functionality is added to the system. Instead these modifications only allow for easier solutions for some kinds of task.

Retrieve the Last Position

When performing tasks where objects are to be palletized, we might use a command `getLastPosition(name)`. The purpose of this command is to simply return the final coordinates of the position after processing all extensions when it was approached the last time. The purpose of this command is to allow to define other positions in relation to this position. The advantage will be that the position is evaluated only once. All other positions depending on this position need not be evaluated using sensors again.
A typical example would be a palletizing task where objects are to be stacked next to each other. The position of the first object in the box is determined using sensors. At this point we can calculate the positions of all other objects if the dimensions of the objects are known. These positions are computed using the `getLastPosition` command in conjunction with the `TRANS` command.

Transmitting Multiple Positions

Up to now, the robot always moves to the next position in a search and checks if the terminating condition holds. But for some searches it may be necessary to perform some kind of detach motion before moving to the next position in the search. An example would be a blind search where the robot shall not move along the surface of the object in order to avoid scratches on the surface. Here, the robot shall only move to specific points on the surface to check the terminating condition.
This results in a specific trajectory from one position in the search to the next. To allow for this feature, the position database should be able to transmit a whole array of positions describing this trajectory. The robot then moves along this array and only informs the database when the final position has been approached.

8.3.2 Complex Sensor Transformations

Due to the abstract nature of this framework the developer must at least parametrize existing sensor transformation functions to his needs. In the worst case, he must design these functions completely on his own. To encounter this problem, two approaches are conceivable:
Firstly, special libraries with sensor transformation functions for certain tasks or sensors can be created by robot and sensor manufacturers. These are tailored to specific problems and alleviate the developer from the task of creating these functions himself. So one option for future work would be to investigate if there are domains of tasks or sensors for which special sets of sensor transformation functions can be encapsulated into libraries that only need to be parametrized. This includes the integration of skills and skill primitives, since these represent geometric change functions on a very abstract level. Another option is to explore if and how abstract skills can be integrated into this framework.
Secondly, learning algorithms may be employed and tailored to the task at hand. The idea here is to let the robot learn complex sensor transformations by himself (maybe with super-

vision by the developer) instead of falling back on existing libraries and functions. While this approach will allow the robot to become even more flexible and adaptive, the downside is that it is hard to employ universal learning algorithms without detailed knowledge of their working. Nonetheless, we believe that this approach is worth some thought as well, since it will provide the developer with a more intuitive way of programming the robot than resorting to a library, as this has to be created by someone as well.

8.3.3 Trajectory Centered Tasks

As discussed in the Section 8.2, tasks that require the robot to modify trajectories based on sensor information can only be solved using workarounds with our approach. In order to increase the range of applications, it should be examined how our framework can be combined with approaches focussing on trajectory centered tasks. There are two problems here:
Firstly, trajectories that are modified by external sensors will require sensor data processing within the robot program since this cannot be done in the position database. Not only will this weaken the desired separation of robot instructions and sensor data processing, but there will be two areas where sensor information is processed: In the robot program (for trajectories) and in the position database (for positions).
Secondly, it must be evaluated if existing approaches to such tasks, like the task frame formalism, can be integrated into our framework without modifications.
The main issue here is to maintain intuitivity for non-experts. In order to allow for the modifications of trajectories in addition to positions, this aspect must have the highest priority.

Bibliography

[1] Abb robotics. http://www.abb.de/robotics.

[2] Adept technologies. http://www.adept.de.

[3] Camelia. http://camellia.sourceforge.net.

[4] Halcon. http://www.mvtec.com/halcon/.

[5] Icra09 workshop: Formal methods in robotics and automation. http://web.mae.cornell.edu/hadaskg/icra09/.

[6] Kuka robotics. http://www.kuka-robotics.com/.

[7] Microsoft robotics studio. http://msdn.microsoft.com/de-de/library/bb483065.aspx.

[8] Qt. http://qt.nokia.com/products.

[9] Smerobot. http://www.smerobot.org/02_overview.

[10] Stäubli robotics. www.staubli.com/de/robotik/.

[11] Lego mindstorms nxt, 2009. http://www.nxt-in-der-schule.de/lego-mindstorms-education-nxt-system/nxt-software/nxt-education-software/nxt-education-software.

[12] Kriesten A. and M. Rößler. Generalisierte plattform zur sensordatenverarbeitung. In *Dresdner Arbeitstagung Schaltungs- und Systementwurf*, 2006. http://www.eas.iis.fhg.de/events/workshops/dass/2006/dassprog/pdf12_kriesten.pdf.

[13] S. Calinon S. Schaal A. Billard, Y. Epars and G. Cheng. Discovering optimal imitation strategies. In *Robotics and Autonomous Systems*, volume 47, pages 65–185, 2004.

[14] Jürgen Acker. *Handhabung deformierbarer linearer Objekte basierend auf Kontaktzuständen und optischer Sensorik*. Shaker Verlag, 2008.

[15] M. Adams. *Sensor Modelling, Design and Data Processing for Autonomous Navigation*. World Scientific Publishing, 1998. ISBN 9810234961.

[16] Martin Anthony and Peter L. Bartlett. Function learning from interpolation, 1994.

[17] D. Bara. Issues in computing contact forces for non-penetrating rigid bodies. In *Algorithmica*, volume 10, pages 292–352, 1993.

[18] Michael Beetz, Alexandra Kirsch, and Armin Müller. RPL-LEARN: Extending an autonomous robot control language to perform experience-based learning. In *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*, 2004.

[19] Robert Bicker, Zhongxu Hu, and Kevin Burn. A self-tuning fuzzy robotic force controller. 2002.

[20] Geoffrey Biggs and Bruce Macdonald. A survey of robot programming systems. In *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, page 27, 2003.

[21] R. Bischoff, A. Kazi, and M. Seyfarth. The morpha style guide for icon-based programming. In *Proceedings 11th IEEE International Workshop on Volume Robot and Human Interactive Communication*, pages 482 – 487, 2002.

[22] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995. ISBN 0-19-853849-9.

[23] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.

[24] George Box, Gwilym M. Jenkins, and Gregory Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, 1994.

[25] A. Breckweg, C. Meyer, K. Drechsler, and O. Rüger. Fast and intuitive robot programming for mandrel guiding of braiding machines for textile preforming. 2007.

[26] Eli Brookner and Joseph J. Cynamon. Book review : Tracking and kalman filtering made easy. *Simulation*, 75(3):170, 2000.

[27] C.G. Broyden. A class of methods for solving nonlinear simultaneous equations. In *Mathematics of Computation*, volume 19, pages 577–593, 10 1965.

[28] Herman Bruyninckx and Joris De Schutter. Specification of force-controlled actions in the task frame formalism: A synthesis. 1999.

[29] Horst Bunke, Takeo Kanade, and Hartmut Noltemeier, editors. *Modelling and Planning for Sensor Based Intelligent Robot Systems [Dagstuhl Workshop, October 24-28, 1994]*. World Scientific, 1995.

[30] S. Calinon and A. Billard. A probabilistic programming by demonstration framework handling constraints in joint space and task space. In *International Conference on Intelligent Robots and Systems*, 09 2008.

[31] John Canny and Eric Paulos. Informed peg-in-hole insertion using optical sensors. In *SPIE Conference on Sensor Fusion VI*, 1993.

[32] P. Cheng, D. Cappelleri, B. Gavrea, and V. Kumar. Planning and control of meso-scale manipulation tasks with uncertainties. In *Robotics: Science and Systems*, 2007.

[33] Siddharth R. Chhatpar. *Localization for Robotic Assemblies with Position Uncertainty*. Case Western Reserve University, 2005.

[34] L.H. Chiang, E.L. Russell, and R.D. Braatz. *Fault Detection and Diagnosis in Industrial Systems*. Advanced Textbooks in Control and Signal Processing. Springer Verlag, 2001.

[35] Christopher Parlitz Martin Hägele Christian Meyer, Rebecca Hollmann. Programming by demonstration for assistive systems - intuitive programming of welding and gluing trajectories. In *it - Information Technology*, volume 49, pages 238 – 245, 2007.

[36] Katja. Dauster. *Prozessangepasste, lernende Roboterregelung für Montageprozesse /*. Düsseldorf : VDI-Verl.,, 2002.

[37] Vasek Chvátal David L. Applegate, Robert E. Bixby and William J. Cook. *The Traveling Salesman Problem. A Computational Study*. Princeton University Press, 2007.

[38] J. De Schutter and H. van Brussel. Compliant robot motion: I. a formalism for specifying compliant motion tasks. *Int. J. Rob. Res.*, 7(4):3–17, 1988.

[39] J. Deiterding and D. Henrich. Automatic optimization of adaptive robot manipulators. In *Proceedings of the 2007 IEEE International Conference on Intelligent Robots and Systems, San Diego/USA*, 2007.

[40] J. Deiterding and D. Henrich. Acquiring change models for sensor-based robot manipulation. In *Proceedings of the 2008 IEEE International Conference on Robotics and Automation, Pasadena/USA*, 2008.

[41] J. Deiterding and D. Henrich. Workpiece drift recognition and adaptation for robot manipulation tasks. In *Proceedings of the 17th International Workshop on Robotics in Alpe-Adria-Danube Region, Ancona/Italy*, 2008.

[42] J. Deiterding and D. Henrich. Online calibration of one-dimensional sensors for robot manipulation tasks. In *Icinco - 6th international conference on informatics in control, automation and robotics 2009, Milan/Italy*, 2009.

[43] J. Deiterding and D. Henrich. Probability based robot search paths. In *German Workshop on Robotics, Braunschweig/Germany*, 2009.

[44] Rüdiger Dillmann. Building elementary robot skills from human demonstration. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 2700–2705, 2003.

[45] Kevin R. Dixon, Martin Strand, and Pradeep K. Khosla. Predictive robot programming. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[46] M. Dong, L. Tong, and B.M. Sadler. Information retrieval and processing in sensor networks: deterministic scheduling vs. random access. In *Proc. o.t. Int. Symp. on Information Theory*, 2004.

[47] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2 edition, November 2000.

[48] G. Dudek and C. Zhang. Vision-based robot localization without explicit object models. 1996.

[49] Bernd Finkemeyer. *Robotersteuerungsarchitektur auf der Basis von Aktionsprimitiven*. Shaker Verlag, 2004.

[50] R.J. Firby. *Adaptive execution in complex dynamic worlds*. Yale university, 1989.

[51] Rudolf Fleischer, Tom Kamphans, Rolf Klein, Elmar Langetepe, and Gerhard Trippen. Competitive online approximation of the optimal search ratio. *SIAM J. Comput.*, 38(3):881–898, 2008.

[52] United Nations Economic Commission for Europe (UNECE). Solid recovery of sales and production of industrial robots in germany - the world's second largest user and producer of industrial robots. In *World Robotics 2004*, 2004.

[53] Douglas W. Gage. Randomized search strategies with imperfect sensors. In *In Proceedings of SPIE Mobile Robots VIII*, pages 270–279, 1993.

[54] Ken Goldberg. Icra 2008 workshop on algorithmic automation, 2008.

[55] J.A. Benbrahim H. Gullapalli, V. Franklin. Acquiring robot skills via reinforcement learning. In *IEEE Control Systems Magazine*, volume 14, pages 13–24, 1994.

[56] G.D. Hager. *Task-Directed Sensor Fusion and Planning: A Computational Approach*. Kluwer, May 1990.

[57] T. Hasegawa, T. Suehiro, and K. Takase. A model-based manipulation system with skill-based execution in unstructured environments. In *Fifth International Conference on Advanced Robotics (ICAR91)*, pages 970 – 975, 1991.

[58] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, Mass., 1995.

[59] R. Hollmann, E. Westkämper, and A. Verl. Industrieroboter schneller und einfacher programmieren : Auch ohne brain-to-robot-interface. *Fraunhofer IPA Workshop F 157*, 1:93–103, 2007.

[60] Alston S. Householder. Unitary triangularization of a nonsymmetric matrix. *J. ACM*, 5:339–342, 1958.

[61] Yanrong Hu and Simon X. Yang. A knowledge based genetic algorithm for path planning of a mobile robot. In *ICRA*, pages 4350–4355, 2004.

[62] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, pages 159–187. Springer Verlag, 2002.

[63] S.A. Hutchinson, R.L. Cromwell, and A.C. Kak. Planning sensing strategies in a robot work cell with multi-sensor capabilities. In *Proc. IEEE Int. Conf. On Robotics and Automation*, pages 1068–1075, 1988.

[64] Verein Deutscher Ingenieure. Vdi 2860: Assembly and handling; handling functions, handling units; terminology, definitions and symbols. pages 1–16, 1990.

[65] H. Kirnura J. Takamatsu and K. Ikeuchi. Classifying contact states for recognizing human assembly task. In *Proceedings IEEE/SICE/RSJ International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 177 – 182, 1999.

[66] Stefano Caselli Jacopo Aleotti and Monica Reggiani. Toward programming of assembly tasks by demonstration in virtual environments. In *12th IEEE Workshop Robot and Human Interactive Communication*, 11 2003.

[67] J. W. Jeon, S. Park, and S. Kim. Compensation for servo drift in industrial robots. In *Industrial Electronics, Control, Instrumentation, and Automation - IECON*, volume 2, pages 589–594, 1992.

[68] Soller K. and Henrich D. Intuitive robot programming of spatial control loops with linear movements. In Friedrich M. Wahl Torsten Kröger, editor, *GWR09 German Workshop on Robotics*, 2009.

[69] Björn Kahl. *Virtuelle Roboterprogrammierung basierend auf einer Any-time fähigen Simulation deformierbarer linearer Objekte*. Shaker Verlag, 2007.

[70] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[71] P. Kesavan and J. H Lee. Diagnostic tools for multivariable model-based control systems. In *Ind. Eng. Chem. Res*, volume 36, pages 2725–2738, 1997.

[72] M. Khatib. *Sensor-based motion control for mobile robots*. 1996.

[73] Alexandra Kirsch. Towards high-performance robot plans with grounded action models: Integrating learning mechanisms into robot control languages. In *ICAPS Doctoral Consortium*, 2005.

[74] Alexandra Kirsch. *Integration of Programming and Learning in a Control Language for Autonomous Robots Performing Everyday Activities*. PhD thesis, Technische Universität München, 2008.

[75] T. Kröger, B. Finkemeyer, and F. Wahl. Compliance and force control for computer-controlled manipulators. *Proceedings of the IEEE-RSJ International Conference on Robotics and Automation*, 1:3011–3016, 2006.

[76] T. Kröger, B. Finkemyer, and F. Wahl. Compliant robot motion: The task frame formalism revisited. In *Journal of Robotics and Mechatronics*, volume 3, pages 1029–1034, 2004.

[77] Steven M. LaValle. Robot motion planning: A game-theoretic foundation. *Algorithmica*, 26(3-4):430–465, 2000.

[78] Tine Lefebvre, Herman Bruyninckx, and Joris De Schutter. Autonomous execution of force-controlled robot tasks. 2007.

[79] Ulf Leonhardt and Jeff Magee. Multi-sensor location tracking. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 203–214, New York, NY, USA, 1998. ACM.

[80] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathmatics*, II(2):164–168, 1944.

[81] F.J. Lopes. Optimizing and automatic deburring system for the glass industry. In *Master Thesis, University of Coimbra*, 2008.

[82] T. Lozano-Perez. Robot programming. In *Proceedings of the IEEE*, volume 71, pages 821–841, 07 1983.

[83] Reyes Rios-Cabrera Jorge Corona-Castuera Mario Pena-Cabrera, Ismael Lopez-Juarez. Machine vision approach for robotic assembly. In *Assembly Automation*, volume 25, pages 204 – 216. Emerald Group Publishing Limited, 2005.

[84] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[85] K. Marti. Path planning for robots by stochastic optimization methods. In *Journal of Intelligent and Robotic Systems*, volume 22, pages 117 – 127, 1998.

[86] Matthew Mason. Compliance and force control for computer-controlled manipulators. *IEEE Trans on Systems, Man, and Cybernetics*, 11(1):418–432, 1981.

[87] David D. Morrison. Remarks on the unitary triangularization of a nonsymmetric matrix. *J. ACM*, 7(2):185–186, 1960.

[88] James Morrow and Pradeep Khosla. Manipulation task primitives for composing robot skills. In *IEEE International Conference on Robotics and Automation (ICRA '97)*, volume 4, pages 3354–3359, 04 1997.

[89] H. Mosemann. *Beiträge zur Planung, Dekomposition und Ausführung von automatisch generierten Roboteraufgaben*. Shaker Verlag, 2000. ISBN 3-8265-7710-8.

[90] W. Neubauer, M. Moller, S. Bocionek, and W. Rencken. Learning systems behaviour for automatic correction and optimization of off-line robot programs. In *Proceedings of the 1992 lEEE/RSJ International Conference on Intelligent Robots and Systems, 1992*, volume 2, page 1355 ? 1362, 07 1992.

[91] D. Orin and W. Schrader. Efficient jacobian determination for robot manipulators. In M. Brady and P. R.P. Paul, editors, *Rob. Research: The 1st int. symposium*. MIT Press, Cambridge, 1984.

[92] Giuseppe De Nicolao Paolo Magni, Riccardo Bellazzi. Bayesian function learning using mcmc methods. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 20, pages 1319–1331, Dec 1998.

[93] N. Paragios and G. Tziritas. Adaptive detection and localization of moving objects in image sequences. *Signal Processing: Image Communication*, 14:277–296, 1999.

[94] M. Pardowitz, R. Zöllner, and R. Dillmann. Incremental acquisition of task knowledge applying heuristic relevance estimation. *IEEE Trans on Systems, Man, and Cybernetics*, 11(1):418–432, 2006.

[95] Parzen and Fabian Hoti. On estimation of a probability density function and mode. volume 33, 1962.

[96] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *In International Conference on Robotics and Automation*, pages 1144–1151, 1999.

[97] Rolf Pfeifer and Christian Scheier. From perception to action: The right direction? In *Proc. "From Perception to Action" Conference*, pages 1–11. IEEE Computer Society Press, Los Alamitos, 1994.

[98] J.N. Pires. New challenges for industrial robotic cell programming. In *Industrial Robot*, volume 36, 2009.

[99] W.H. Press, B.P. Flannery, S.A. Teukolsky, and Vetterling W.T. Secant method, false position method, and ridders' method. In *Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed.*, pages 347–352. Cambridge University Press, 1992.

[100] M. Ehrenmann O. Rogalla R. Dillmann, R. Zöllner. Interactive natural programming of robots: Introductory overview. In *Proc. of DREH 2002*, 2002.

[101] Raul Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1 edition, July 1996.

[102] K. Rui, M. Yoshifumi, and M. Satoshi. Information retrieval platform on sensor network environment. In *IPSJ SIG Technical Reports*, number 26, pages 37–42, 2006. ISSN 0919-6072.

[103] F. Guenter S. Calinon and A. Billard. On learning, representing, and generalizing a task in a humanoid robot. In *IEEE Trans. Syst., Man, Cybernetics*, volume 37 of *B*, pages 286–298, 2007.

[104] Antoine Schlechter. *Einhändige kraftbasierte Handhabung deformierbarer linearer Objekte*. Shaker Verlag, 2007.

[105] Antoine Schlechter and Dominik Henrich. Manipulating deformable linear objects: Manipulation skill for active damping of oscillations. In *Proceedings of the 2002 IEEE International Conference on Intelligent Robots and Systems, Lausanne/Switzerland*, 2002.

[106] Antoine Schlechter and Dominik Henrich. Discontinuity detection for force-based manipulation. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation, Orlando/Florida*, 2006.

[107] M. Schlemmer and G. Grübel. Real-time collision- free trajectory optimization of robot manipulators via semi-infinite parameter optimization. In *The International Journal of Robotics Research*, volume 17, pages 1013–1021, 1998.

[108] R. D. Schraft and Christian Meyer. The need for an intuitive teaching method for small and medium enterprises. In *ISR 2006 - ROBOTIK 2006 : Proceedings of the Joint Conference on Robotics*, May 2006.

[109] Rajeev Sharma, Steven M. Lavalle, and Seth Hutchinson. Optimizing robot motion strategies for assembly with stochastic models of the assembly process. *IEEE Trans. on Robotics and Automation*, 12:145 – 157, 1996.

[110] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer, Berlin, Heidelberg, 2008.

[111] David Simon, Lee Weiss, and Arthur C Sanderson. Self-tuning of robot program primitives. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation (ICRA '90)*, volume 1, pages 708 – 713, May 1990.

[112] R. Suarez, L. Basanez, and J. Rosell. Using configuration and force sensing in assembly task planning and execution. *Assembly and Task Planning, IEEE International Symposium on*, 0:0273, 1995.

[113] Phillip D. Summers and David D. Grossman. Xprobe: An experimental system for programming robots by example. In *International Journal of Robotics Research*, volume 3, 1984.

[114] Aaron Tenenbein and Jay-Louise Weldon. Probability distributions and search schemes. *Information Storage and Retrieval*, 10(7-8):237–242, 1974.

[115] Ulrike Thomas. *Automatisierte Programmierung von Robotern für Montageaufgaben.* Shaker Verlag, 2005.

[116] Ulrike Thomas, Bernd Finkemeyer, Torsten Kröger, and Friedrich M. Wahl. Error-tolerant execution of complex robot tasks based on skill primitives. In *In IEEE International Conference on Robotics and Automation*, pages 3069–3075, 2003.

[117] Ulrike Thomas, Aanton Movshyn, and Friedrich Wahl. Autonomous execution of robot tasks based on force torque maps. In *Proceedings of the Joint Conference on Robotics. International Symposium on Robotics / Robotik 2006*, 2006.

[118] Sebastian Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA*, pages 306–312. IEEE. http://robots.stanford.edu/papers/thrun.ces-icra.html, 2000.

[119] Robert Tilove. Smart assembly: Industrial needs and r&d challenges. 2008. http://www.give.nl/events/ICRA_AA/index.html.

[120] Bertrand Tondu. The three-cubic method: An optimal online robot joint trajectory generator under velocity, acceleration, and wandering constraints. In *The International Journal of Robotics Research*, volume 18, pages 893–901, 1999.

[121] Christian Meyer und Olaf Rüger. Intuitive programmierung von robotern beim flechten von kohlefasern. In *Maschinenmarkt*, 2007. http://www.maschinenmarkt.vogel.de/index.cfm?pid=1576&pk=95219.

[122] Naumann M. Verl, A. Plug and produce-steuerungsarchitektur für roboterzellen: Automatische code-generierung für roboterzellen aus prozess- und gerätebeschreibungen. In *Wt Werkstattstechnik*, volume 98, pages 384–390, 2008.

[123] VDI Nachrichten vom 19.11.2004. Die automatisierungsbranche befindet sich im wandel - studie 'automatisierung 2010'. *VDI Nachrichten*, 19.11. 2004.

[124] P. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. John Wiley & Sons, New York, 1994.

[125] M.D. Wheeler. Automatic modeling and localization for object recognition. 1996.

[126] Alexander Winkler. *Ein Beitrag zur kraftbasierten Mensch-Roboter-Interaktion*. 2006. Dissertation.

[127] Jing Xiao. Automatic determination of topological contacts in the presence of sensing uncertainties. In *Proc. 1993 IEEE International Conference on Robotics and Automation*, volume 1, pages 65–70, 05 1993.

[128] Jing Xiao and Lianzhong Liu. Contact states: Representation and recognizability in the presence of uncertainties. In *in the Presence of Uncertainties ", IEEE/RSJ Int. Conf. Intell. Robots and Sys*, 1998.

[129] T. Yoshikawa and K. Yoshimoto. Haptic simulation of assembly operation in virtual environment. In *Proceedings of the ASME Dynamic Systems and Control Division*, volume 69, pages 1191–1198, 2000.

[130] Shigang Yue and Dominik Henrich. Manipulating deformable linear objects: Sensor-based skills of adjustment motions for vibration reduction. In *Journal of Robotic Systems 22(2), 67-85 (2005)*, 2005.

[131] Shigang Yue and Dominik Henrich. Manipulating deformable linear objects: Fuzzy-based active vibration damping skill. In *Journal of Intelligent Robot Systems 46/2006, pp. 201-219*, 2006.

[132] A. Zelinsky, R.A. Jarvis, J. C. Byrne, and S. Yuta. Planning paths of complete coverage of an unstructured environment by a mobile robot. In *In Proceedings of International Conference on Advanced Robotics*, pages 533–538, 1993.