A Generic Framework for Robot Motor Planning and Control

S A G A R   B E H E R E

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2010

A Generic Framework for Robot Motor Planning and Control

S A G A R   B E H E R E

Master's Thesis in Computer Science (30 ECTS credits)
at the Systems, Control and Robotics Master's Program
Royal Institute of Technology year 2010
Supervisors at CSC were Patric Jensfelt and Christian Smith
Examiner was Danica Kragic

TRITA-CSC-E 2010:014
ISRN-KTH/CSC/E--10/014--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

To my parents,
who didn't always understand
but always accepted and supported.

Abstract

This thesis deals with the general problem of robot motion planning and control. It proposes the hypothesis that it should be possible to create a generic software framework capable of dealing with all robot motion planning and control problems, independent of of the robot being used, the task being solved, the workspace obstacles or the algorithms employed. The thesis work then consisted of identifying the requirements and creating a design and implementation of such a framework. This report motivates and documents the entire process. The framework developed was tested on two different robot arms under varying conditions. The testing method and results are also presented. The thesis concludes that the proposed hypothesis is indeed valid.

Keywords: path planning, motion control, software framework, trajectory generation, path constrained motion, obstacle avoidance.

Referat

Ett generellt ramverk för rörelseplanering och styrning för robotar

Denna avhandling behandlar det generella problemet att planera och reglera rörelsen av en robot. Arbetshypotesen är att det är möjligt att skapa ett ramverk som kan hantera alla rörelseplanerings- och reglerproblem, oberoende av vilken robot som används, uppgift som skall utföras, arbetsområdets beskaffenheter eller de algoritmer som används. Examensarbetet bestod i att identifiera behov och skapa en design för och implementera ett sådant ramverk. Denna rapport motiverar och dokumenterar hela processen. Ramverket har testats på två dolika robotarmar under olika förhållanden. Testmetod och resultat från dessa tester presenteras. Exjobbets slutsats är att den föreslagna hypotesen gäller.

Acknowledgements

Pride of place goes to Patric Jensfelt. Words fail me if I try to express the immense gratitude I feel towards him. A *guru* in every sense of the word, Patric has been my thesis supervisor, professor, Master program director and the ultimate solution to all problems. My time in Sweden has been fun and interesting due to the opportunities he provided.

Christian Smith answered questions as fast as I could ask them and often, faster. Learning from him is like drinking from a firehose. He also made available the Powercube robot arm for me to test my theories on.

Mitul Saha provided support for the Motion Planning Kit used in the thesis. He even changed the licensing terms to better suit my requirements! Motion planning is a critical part of the thesis and Mitul's responsiveness gave me a lot of reassurance.

Jeannette Bohg planted the seed of the problem in my mind. Without her, I would be writing a thesis on a completely different topic.

Alper 'the Bhai' Aydemir uncomplainingly bore my rantings on Life, the Universe and Everything. I utilized many of the technical ideas and approaches that seem to float around him like a cloud.

Xavi Gratal Martinez was the single point source of support for the KUKA robot arm and Schunk Hand used during the thesis.

The thesis was carried out with the infrastructure provided by the Center for Autonomous Systems (CAS) at KTH. The institute's role is gratefully acknowledged.

Contents

Chapter 1

Introduction

Robots have the potential to improve efficiency, safety and convenience of human endeavors. To realize this potential, research in robotics necessarily involves investigation into a broad range of disciplines. A fundamentally important discipline is that of motion planning and control. It is important because motion as a requirement is common to all robots. Robots need to move in order to do something useful, and this holds true regardless of whether the robot is mobile or fixed, autonomous or non-autonomous.

While the fact remains that all robots need to move, the actual motion executed is highly dependent on the robot under consideration, the task it is fulfilling and the constraints it operates under. Therefore, numerous concepts, theories and algorithms have been developed for solving specific classes of motion problems.

Considering the multitude of robots, theories and applications, the solutions of robot motion tasks are often surprisingly narrow. Narrowness implies that the same solution is rarely used for solving a problem different from what it was immediately designed for. This is surprising because adapting an existing solution to a similar problem involves, in general, a lower quantum of work than solving the problem all over again.

This thesis is a step towards creating a universal solution through which knowledge of motion planning and control can be applied to a robot motion task, regardless of the task, the robot or the theory involved.

The abstract notion has now been introduced. The rest of this chapter makes it progressively more concrete.

1.1 The motion planning and control problem

The locus of points along which the robot needs to move in order to go from one point to another is called the path. Determining the path can be a non-trivial problem, especially if obstacles are present in the robot's environment. Obstacles are any physical objects in the environment which can potentially impede the robot's motion. Thus, given a destination to be reached, the core problem of motion planning involves finding a path from the current robot location to the destination without colliding with obstacles, if any. The core problem can be advanced by placing additional demands on the characteristics of the path. Characteristics of the path refer to the velocities/accelerations of the robot along the path, the energy expended in moving along the path, the time needed for completing the motion et cetera.

Once a suitable path has been found, the robot actuators should generate the correct forces/torques at all instants, so that the robot moves along the path. This is the control problem.

Typically, robots are designed to fulfill specific tasks and appropriate algorithms for planning and control are incorporated in the robot control software. Often, high level planning algorithms are absent and the robot control software merely drives the robot according to programmed motion commands. In such a case, the robot programmer is responsible for specifying the proper paths to be followed.

1.2 Motivation for the master thesis

It is not uncommon in robotics for the same solution to get implemented many times over. Some reasons are

- Existing implementations are tied down tightly to a specific robot model
- The implementations are owned by proprietary companies and thus not freely available for others to use
- An implemented algorithm depends on a host of other software services from which it cannot be easily extracted for generic use. Hence, software developers often examine existing code and then write their own version
- The diversity of robots, control techniques and user interfaces makes it difficult to develop a 'one size fits all' type of solution. It is simply more convenient to narrow down the requirements and generate a tailor made solution for them

The repeated solutions to similar problems consume effort and creativity that could be applied to solving more original problems. The motivation for this master thesis arose from pursuing a "What if ... ?" line of thought

- What if a single software could be used to control motion of any robot?
- What if the same software could be used for any robot motion task?
- What if the same software could execute any desired algorithm(s) to solve the task?
- What if the same software could be used to provide a set of 'robot motion services' to interested parties?
- What if the software was free, open, extensible with a growing set of supported robots, algorithms and areas of application?

This is a good scenario for applying the software framework paradigm. In a software framework, common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality[22]. The specialized user code in this case would be the code for interacting with a particular robot, or a specific path planning algorithm that needs to be applied to the problem being solved and so on.

There are numerous advantages to the framework approach. In a scenario like a robotics research lab, deployment of the framework can provide researchers with a uniform interface to all the lab robots. Code can be written to interact with the framework, instead of a specific robot. Thus, the code would then be immediately applicable to all similar robots which the framework supports. The same algorithm can be tested on different robots, or different algorithms can be quickly run on the same robot. Quirks of manufacturer specific software need not influence the interaction with the robot. The framework services can also be used for motion sub-problems like finding collision free paths under different conditions, solving inverse kinematics, moving the robot along specified paths etc. The configurability of the framework also implies that the same generic software can be used to create a solution highly tuned to a specific problem.

A potential risk in the creation of a generic solution lies in the possibility that some design decisions may not be the most optimal for a certain problem. In such cases, a tailor made solution would be more appropriate. However, the existence of this risk does not negate the importance or utility of a generic solution. The presence of a generic solution does not imply an obligation to apply it, while its absence will not improve the situation. Other problems continue to exist, to which the generic solution is satisfactorily applicable.

A generic motion planning and control framework has the potential to be advantageous for a wide variety of robot use cases. Realizing this potential is sufficient motivation for pursuing the development of such a framework as a Master thesis.

1.3 Contribution and outline of the thesis

The working hypothesis throughout the thesis is that it is possible to have a generic software framework for robot motion planning and control, which can be configured to solve specific robot motion tasks. The contribution of this thesis is the validation of the hypothesis via realization and real-world testing of such a framework. The realized framework is also made available to the robotics community as a ready-to-use 'product'. Additionally, the thesis work also resulted in the creation of tools to test the realized framework and the development of a software library which can be used by software programs to interact with the framework.

Chapter 2 of this thesis report examines the current state of the art by looking at existing solutions which can directly or indirectly be used for validating the hypothesis. Chapter 3 presents a quick overview of the theory needed to tackle a motion planning and control problem. Chapter 4 discusses the requirements the framework must fulfill to achieve its functional goals. Chapter 5 discusses the implementation of a specific framework which was developed for validating the hypothesis. Chapter 6 describes how the implementation was tested and presents a real-world use case. Chapter 7 concludes the report and discusses possible future work.

1.4 Terminology

Some terms used in this report have a specific meaning in the context of robot motion planning and control. They are described here

Robot configuration A complete specification of the position of every point of the physical robot. Usually described by the set of values of all the robot degrees of freedom

Configuration space The space of all possible robot configurations

Joint space Same as configuration space (Used for kinematic chain robot manipulators)

Cartesian space The normal Euclidean space in which the robot operates

Operational space Same as Cartesian space

Forward kinematics A mapping from configuration space to Cartesian space

Inverse kinematics A mapping from Cartesian space to configuration space

Sarathi	The name of the robot motion planning and control framework developed as part of this thesis work
Parth	The name of a software tool used for scene visualization and testing of Sarathi

Chapter 2

Current state of the art

This thesis is about the design and development of a generic software framework for robot motion planning and control. Thus, a survey of the current state of this art involves a discussion of existing software that also aspires towards a similar goal. The focus is not on solutions to motion planning and control, nor on generic software frameworks. The art in this context refers to the specific use of software frameworks to create a generic solution applicable to any motion planning and control problem. A fairly rigorous search turned up only two other software that could qualify for being generic motion planning and control frameworks. These are OpenRAVE[15] and RobWork[19].

OpenRAVE is an open-source, cross-platform architecture for integration and testing of high-level scripting, motion planning, perception and control algorithms. Targeted for autonomous robotic applications, it includes a seamless integration of 3-D simulation, visualization, planning, scripting and control. A plugin architecture allows users to easily write custom controllers or extend functionality. OpenRAVE focuses on autonomous motion planning and high-level scripting rather than low-level control and message protocols. For the latter purposes, it includes interfaces to other popular robotics packages like Player[17] and ROS[20]. OpenRAVE addresses a superset of the general motion planning and control problem. It is like a 'Do-It-Yourself' kit providing software building blocks for generating a specific application.

RobWork is a framework for simulation and control of robots with emphasis on industrial robotics and its applications. Its major goal is to provide a single framework for offline and online robot programming including modelling, simulation and (realtime)control of robots. A separate application called RobWorkStudio is available for visualization of the robot and its workcell. There are some startling similarities between RobWork and Sarathi, the framework developed during this thesis. Unfortunately, the work on both software was carried out in isolation and the first releases were made almost simultaneously, at which point the developers became aware of the existence of the other software. Despite the similarities, some important differences still exist in the way RobWork and Sarathi have been implemented. RobWork attempts to define uniform global data structures for robots, robot data and workcells. It includes classes for defining vectors, kinematic frames, path planners and so on. Thus it appears that all the concepts within RobWork are tightly integrated with each other. Although the RobWork description refers to it as a 'framework', it exists as a collection of C++ libraries and is referred to as such in its documentation. Sarathi, on the other hand is not a library. It exists as a framework, providing a set of services. Client programs need to connect to Sarathi to use its services. The components in Sarathi are loosely coupled. There is no uniform representation for the data each component uses internally. Since RobWork exists as a libary, it should be possible to use its features for writing plugins for Sarathi. Thus, RobWork can potentially be used for rapidly extending Sarathi.

Chapter 3

Motion planning and control: A theoretical overview

A practical solution of a motion planning and control problem involves solving several sub-problems. This chapter gives an introduction to the main sub-problems, which are

1. Path planning
2. Trajectory generation
3. Motion control
4. Sensing and estimation

Each sub-problem is a subject of extensive research and no attempt is made to provide a comprehensive survey of the field. Rather, the intention is to provide sufficient theoretical knowledge necessary for grasping the solutions which could be implemented in the framework. References are provided to sources of further, in-depth knowledge.

3.1 Path planning

Physical objects in the workspace of a robot present potential obstacles to its motion. Path planning is the process of finding a path which the robot should follow, in order to avoid collisions with these obstacles. A path denotes the locus of points in the robot's configuration space, or in cartesian space, which the robot has to follow in the execution of the assigned motion. Note that a path does not involve the notion of time, it is a purely spatial concept.

An in-depth discussion of robot motion planning is given in [45]. More general information on planning algorithms, with a specific section on motion planning is given in [63].

3.1.1 The path: representation and characteristics

A path is usually represented either as a parametric curve in space or directly as an ordered set of points.

The parametric representation is

$$p = p(u), \qquad u \epsilon [u_{min}, u_{max}] \tag{3.1}$$

where $p(.)$ is a R^6 vector valued function in case the path is in cartesian space[1]. In case the path is in configuration space, $p(.)$ is a $(Nx1)$ vector valued function where N is the dimension of the configuration space. In practice a geometric path is composed of a number of segments, i.e

$$p(u) = p_k(u), \qquad\qquad k = 0, ..., n-1 \tag{3.2}$$

In simple cases, the polynomials p_k describing each segment can be analytically obtained by means of circular/straight line motion primitives. An in-depth treatment of this topic can be found in [48], [75] and [32]. More often, the polynomials must be obtained by using more complex approaches which guarantee continuity of the curve and its derivatives up to a desired order. This continuity is desirable because successive derivatives of a path represent velocity, acceleration, jerk and so on. Discontinuities in these quantities have adverse effects on the robot dynamics and actuators. In most cases, the acceleration needs to be continuous while the jerk may be discontinuous but should be below a specified limit. In case of fast dynamics and high inertias, it is desirable to have continuous jerk as well. The use of a parametric path for trajectory planning usually requires that the path be *geometrically* as well as *parametrically* continuous. Geometric continuity implies that the path is geometrically smooth, while parametric continuity implies that the parametric derivative vectors $\left(\frac{dp}{du}, \frac{d^2p}{du^2}, ...\right)$ are continuous. Two infinitely differentiable segments meeting at a common point, i.e. $p_k(1) = p_{k+1}(0)$[2] are said to meet with *n-order parametric continuity* denoted by C^n, if the first n derivatives match at the common point, that is if

$$p_k^{(i)}(1) = p_{k+1}^{(i)}(0), \qquad\qquad i = 1, ..., n \tag{3.3}$$

Classical approaches for obtaining continuous derivatives are based on B-spline functions, Bézier curves or NURBS[85] which are piecewise polynomial functions defined by

$$p(u) = \sum_{j=0}^{m} p_j B_j(u) \tag{3.4}$$

where p_j are the control points which determine the shape of the curve by weighting the basis functions $B_j(u)$. More details, definitions and significant properties of B-spline, Bézier and Nurbs curves with applications to motion control can be found in the appendix of [32].

The output of typical path planning algorithms, however, is rarely in the form of a parametrized curve. Path planning algorithms generally output a path as an ordered sequence of points[3]. The distribution of these points is usually dependent on the distribution of obstacles in the workspace as well as the subset of the workspace through which motion is desired. In general, no assumptions can be made about the distribution of points output by the path planner. In this case, the task of building a parametric representation with continuity of the desired order is left to the trajectory generator. This is because the trajectory generator can perform this task while simultaneously optimizing some motion parameters. Also, generating a continuous curve through the output of the path planner can be done in a variety of ways depending on the task being carried out. All these considerations do not affect the path planning process and hence an ordered set of collision free points is considered an acceptable output from a path planner.

[1](6x1) because we assume a minimal representation of cartesian space which lies in R^6.

[2]It is assumed that $u\epsilon[0,1]\forall k$.

[3]The points can be in configuration or cartesian space, depending on the path planner.

3.1.2 Properties of planning algorithms

[45] characterizes a planning algorithm according to the task it addresses, properties of the robot solving the task and properties of the algorithm. This characterization helps in selecting an appropriate algorithm for the problem under consideration.

The task solved can be either of *navigation, coverage, localization* or *mapping*. Navigation involves finding a collision free path from one robot position to another. Coverage involves passing a sensor over all points in a workspace. Localization is the problem of using sensor data and a map to determine the position of the robot. Mapping refers to constructing a representation of an unknown environment which is useful for the other three tasks. The representation is constructed from data obtained while the robot is moving around, and hence a good path for the motion must be determined. The framework under consideration in this thesis is mostly devoted to the navigation problem.

The effectiveness of a planning algorithm depends heavily on the robot utilizing that plan. This is because the robot characteristics determine the degrees of freedom available to the planner, the topology of the configuration space and the constraints on the robot motion (for example, holonomicity). If the robot is modeled with dynamic equations, the force and torque data can be used to compute paths optimized for these variables.

An important characteristic of the planner is the space it works in. This could either be the robot's operational space (i.e. cartesian space) or the robot's configuration space. The space has an impact on the representation of obstacles. Specifically, the difficulty of representing an obstacle in configuration space[87, 65] increases with the number of degrees of freedom of the robot. A path generated in cartesian space can be directly checked for intersection with workspace obstacles. However, configuration space paths provide an important advantage if the robot motion can be commanded in configuration space. This is because the problem of motion singularities can be avoided or easily resolved in configuration space.

A planner can simply generate a path that satisfies all constraints or it can additionally optimize some parameter(s) while generating the path. Commonly optimized parameters are the path length, the energy consumption while following the path and the execution time for the motion.

Computational complexity is another property of the path planner. Computational complexity gives an indication of how much the planner performance will degrade when the inputs are scaled up. The inputs could be the degrees of freedom of the robot, the number of obstacles etc., while the planner performance can be measured in the running time, memory requirements and so on. The performance can be a constant or a polynomial or exponential function of the input size. A planner is often considered practical only if it runs in polynomial time or better with respect to input size [45].

A very desirable property of path planners is completeness. A planner is complete if it can always find a solution (if one exists) or indicate failure in finite time. As the degrees of freedom of a robot increase, complete solutions become computationally intractable. Hence, weaker forms of completeness exist. A planner is resolution complete if it can find a solution in finite time at a given resolution of discretization, if such a solution exists. A planner is probabilistically complete if the probability of finding a solution (if it exists) approaches 1 as time tends to infinity. Probabilistic planning methods have found favor in recent times because they work very well with high-dimensional configuration spaces and their execution speed under these conditions can be several orders of magnitude faster than other known methods.

A planner is considered offline if it constructs the entire plan before motion starts. On the other hand, an online planner incrementally constructs the plan as the robot moves. Online planners may require the use of additional sensors to detect obstacles during motion, and the ability to process that sensor information in real-time. However, with fast computers and feedback loops, the distinction between offline and online planning has blurred. This is because if an offline planner executes quickly enough, it can be used to calculate a new path when a sensor update provides changed data about the environment.

The properties of planners described here generally clash with each other in the sense that performance cannot be simultaneously improved on all fronts. Therefore, depending on the problem being solved appropriate tradeoffs need to be made.

3.1.3 Probabilistic roadmap methods

Planning algorithms, together with their variations and optimizations are legion. A survey of planning algorithms is outside the scope of this report. An entire book devoted to motion planning is [45]. Chapter 12 of [75] provides an overview of popular planning methods for robotics, with a brief discussion of the theory and applicability of each method. This aim of this section is to provide an understanding of the principles behind one of the most popular planning techniques: Probabilistic planning.

Probabilistic planners represent a class of relatively modern methods which work with remarkable efficiency, especially in problems involving high dimensional configuration spaces. They are therefore well suited to solving problems involving generic robotic manipulators (which often have upwards of 5 degrees of freedom). Extensions are available for applying the theory under non-holonomic constraints and thus these planners can be used for motion planning of non-omnidirectional mobile robots as well. The breadth of applications makes these planners worth of study.

Probabilistic planning for robots was developed at many sites [58, 80, 68, 60, 59, 61, 27, 26]. The probabilistic foundations of the method are discussed in [55]. Probabilistic methods fall under the more general category of sampling based methods. A generic survey of sampling based methods is found in [25]. Analysis and path quality of sampling based methods is discussed in [53]. A comparative study of probabilistic roadmap planners can be found in [51].

Probabilistic planners work by determining a finite set of collision free configurations which are used to represent the connectivity of the free configuration space. These configurations are then used to build a roadmap that can be employed for solving motion planning problems. At each iteration of the planner, a random configuration is generated from a sampling of the configuration space. See [52] for an overview of some of the sampling techniques used. The generated configuration is checked for collision (as well as contact) with the workspace obstacles[4]. If the configuration is collision free, it is added to the roadmap and connected to previously stored near[5] configurations if possible. The connection is made by a procedure known as a local planner. The local planner usually constructs a straight line segment in configuration space between the two points in question. The segment is then checked to see if it is collision free. This is usually done by uniformly sampling the segment at a sufficient resolution and checking if the samples are collision free. Adaptive collision checking algorithms also exist which guarantee detection of collisions[73] if they are present. If the segment has collisions, it is discarded and a local path between the points cannot be established.

The planner iterations usually stop after a predetermined number of iterations have been reached. At this stage it is possible to solve the path planning problem by connecting the

[4]Collision checking is, in general, an extremely fast activity.

[5]The nearness metric can be defined in a variety of ways. A common way is to use Euclidean distance in the configuration space.

start and end configurations of the desired motion to the roadmap by collision free local paths. Then a route through the roadmap can be found along free local paths. If a large set of candidate paths become available over time, they can be heuristically pruned down to smaller subsets as described in [36]. Figure 3.1 shows an example roadmap and the solution to a particular problem.

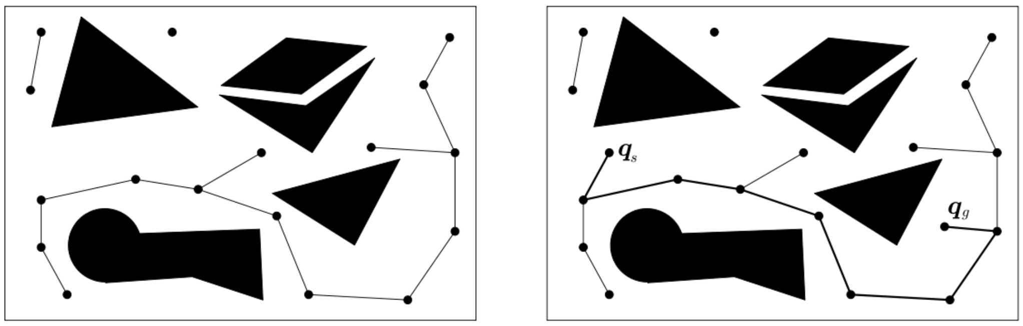


Figure 3.1. A probabilistic roadmap and solution to a particular problem. [75]

If a solution cannot be found, more iterations of the planner can be carried out to generate a more detailed roadmap. Once the roadmap has been sufficiently developed, new instances of the problem can be solved with remarkable speed. Every new query of the planner improves the usage both in terms of connectivity and time efficiency.

Probabilistic roadmap algorithms are simple to implement. In particular, the generation of a configuration space representation of the obstacles is completely avoided.

The basic concept of probabilistic roadmaps can be optimized for further reduction of the time needed to get a solution. An important development is *single-query* probabilistic methods. These methods do not rely on the generation of an exhaustive roadmap of the entire free configuration space. Rather, for each query, a roadmap of only the relevant part of the configuration space is constructed. The bidirectional RRT method is an example of a single query approach. It makes use of a data structure called RRT (Rapidly-exploring Random Tree). More information about path planning with RRTs can be found in [56, 62, 64]. The Single-query Bi-directional Lazy (SBL) planner[72] is another efficient single query planning algorithm which incorporates lazy collision checking [34].

3.2 Trajectory generation

While a path specifies the points[6] in space which should be reached during motion, a trajectory specifies the points and also the time at which those points should be reached during motion. Thus, one can think of a trajectory as a path onto which time constraints are imposed, for example in terms of velocity/acceleration at each point on the path.

Trajectories can be one- or multi-dimensional. A one-dimensional trajectory defines the position for a one degree of freedom system and can be formally defined by a scalar function of the form $q = q(t)$. A multi-dimensional trajectory simultaneously defines the values of multiple degrees of freedom and is formally represented as a vectorial function of time, $\mathbf{p} = \mathbf{p}(t)$. Multi-dimensional trajectories are the norm in robotics, since most robots have more than one degree of freedom.

[6]Note that the points under consideration may be multi-dimensional.

Trajectories are also categorized according to the motion type, which can be point-to-point or multi-point. For the point-to-point case, the desired motion is defined by the initial and final points of motion only. The multi-point case also considers a set of intermediate points which must be reached during the motion.

It is fairly common to combine the tasks of path and trajectory generation. This is almost always the case when obstacles are absent. The concepts developed for point-to-point motion are usually extended to accommodate the case with intermediate points.

The fundamental theory of trajectory planning can be found in books on robotics like [48] and [75]. An excellent book entirely devoted to trajectory planning is [32].

In this section, we take a brief look at a very specific aspect of trajectory planning, that of converting a prespecified multi-dimensional, multi-point geometric path into a trajectory. This is because in the context of a framework for motion planning among obstacles, the path is already generated by the path planner. What remains to be done is generating intermediate points on the path and assigning time values for when those points must be reached. We now consider these two actions separately.

3.2.1 Fitting a set of data points

When a suitable parametric representation of the path is not available, the path must be considered as nothing more than an ordered sequence of points. To generate a suitable curve through the available points, two types of fitting approaches can be distinguished [70, 50]:

Interpolation The curve passes exactly through all points for some value of the independent variable

Approximation The curve need not exactly pass through all points, but passes through their neighborhood within a prescribed tolerance

The approach of choice depends on the problem being solved. For example, approximation is preferred to interpolation when the goal is to construct a curve reproducing the "shape" of the data, avoiding fast oscillations between contiguous points (thus reducing the curvature/acceleration along the trajectory) [32]. Approximation permits the reduction of the speed/acceleration along the curve at the expense of lower precision. Approximation is also the only option when the curve must fit a large number of points, but the free parameters characterizing the curve are insufficient for obtaining an exact interpolation. On the other hand, when the tolerance of approximation is too high and precise motion control is necessary, interpolation is preferred. See figures 3.2 and 3.3 for an example of interpolation and approximation over the same data points. In this section, we will consider interpolation only, since for obstacle avoidance in cluttered workspaces, it is desirable to pass exactly through the points generated by the path planner.

The interpolating/approximating curve can be determined either by global or local procedures. A global procedure considers all the points before generating the trajectory control points. Therefore, if only some points of the path are subsequently modified, the shape of the entire curve changes and the entire trajectory needs to be recalculated. Local procedures are based on local data (tangent vectors, curvature vectors etc.) for each pair of points. These algorithms are computationally less expensive and the resulting curve is easier to modify, but it becomes harder to achieve desired continuity constraints at each point.

The theoretically simplest approach to obtain an expression for a curve fitting N data points is to use an $(N-1)$ order polynomial. However, this approach has severe disadvantages

- It is not possible to enforce initial and final velocities

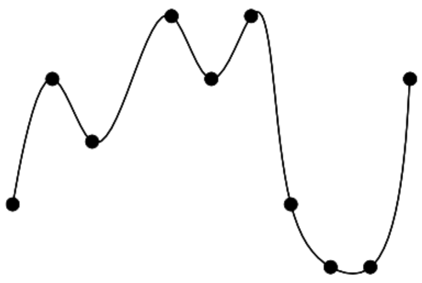


Figure 3.2. Interpolation fitting [32]

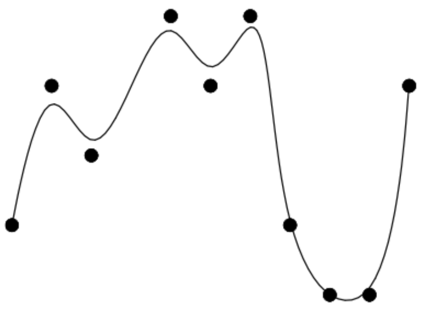


Figure 3.3. Approximation fitting [32]

- High order polynomials can exhibit oscillatory behavior, leading to unnatural trajectories
- As the order of a polynomial increases, the numerical accuracy of their solutions decreases
- Changing a single data point necessitates recalculation of the polynomial

Therefore, the preferred approach is to use a suitable number of low order polynomials with desired continuity constraints at the path points. Generally, at least a cubic polynomial is used since it allows the imposition of continuity on the velocity at each point. The following situations occur commonly and have well-defined mathematical solutions

1. Arbitrary values of velocity are imposed at each data point
2. The velocity values at each data point are assigned according to a specific criterion
3. The acceleration at each data point is constrained to be continuous
4. Linear polynomials are used with parabolic blending at each data point

Further details of each method can be found in chapter 4 of [75]. The basic theory of using splines for computing multipoint trajectories is described in chapter 4 of [32]. The use of algebraic and trigonometric splines is further discussed in [81]. Methods of generating smooth, constrained and time optimal trajectories using splines are presented in [44, 46, 47].

Regardless of the method employed, the ultimate outcome is a path with the desired properties, represented in a parametric form.

3.2.2 Imposition of a timing law

Trajectories generally do not consider system dynamics. Hence, given a geometric path $\mathbf{p} = \mathbf{p}(u)$, the trajectory is completely defined when the timing law $u = u(t)$ is provided. This timing law is specified such that the constraints on velocity and acceleration are met. The timing law is effectively a reparametrization of the path, which modifies the velocity and acceleration vectors, see figure 3.4. Chapter 9 of [32] provides great detail on the imposition of timing laws.

The derivatives of the trajectory $\tilde{\mathbf{p}}(t)$ can be computed by the chain rule on $\tilde{\mathbf{p}}(t) = (\mathbf{p} \circ u)(t)$

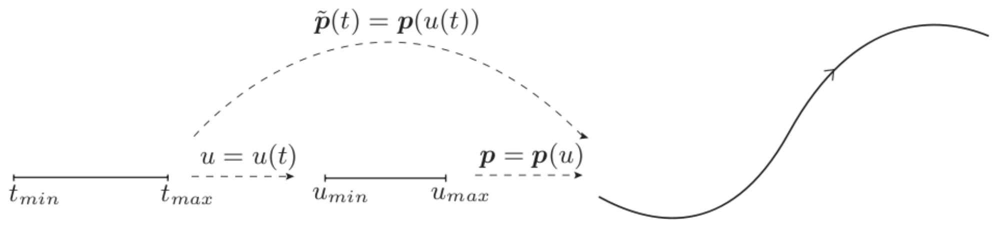


Figure 3.4. Composition of a generic 3D path $\mathbf{p}(u)$ and of a motion law $u(t)$

$$
\begin{aligned}
\dot{\tilde{\mathbf{p}}}(t) &= \frac{d\mathbf{p}}{du}\dot{u}(t) \\
\ddot{\tilde{\mathbf{p}}}(t) &= \frac{d\mathbf{p}}{du}\ddot{u}(t) + \frac{d^2\mathbf{p}}{du^2}\dot{u}^2(t) \\
&\vdots
\end{aligned}
\tag{3.5}
$$

The actual specification of the timing law depends on the problem being solved. A generic motion law simply specifies the relationship between t and u and the derivatives of the parametric curve are calculated according to equation 3.5.

The simplest law is a proportional relation between t and u, i.e $u = \lambda t$. When this is so, the k^{th} derivative of the parametric curve is simply scaled by a factor of λ^k. Thus,

$$
\begin{aligned}
\tilde{\mathbf{p}}^{(1)}(t) &= \frac{d\mathbf{p}}{du}\lambda \\
\tilde{\mathbf{p}}^{(2)}(t) &= \frac{d^2\mathbf{p}}{du^2}\lambda^2 \\
\tilde{\mathbf{p}}^{(3)}(t) &= \frac{d^3\mathbf{p}}{du^3}\lambda^3 \\
&\vdots
\end{aligned}
\tag{3.6}
$$

These relations can be used when the trajectory must satisfy constraints on velocity (v_{max}), acceleration (a_{max}), jerk (j_{max}) etc. In that case, ensuring that

$$\lambda = min\left\{ \frac{v_{max}}{|\mathbf{p}^{(1)}(u)|_{max}}, \sqrt{\frac{a_{max}}{|\mathbf{p}^{(2)}(u)|_{max}}}, \sqrt[3]{\frac{j_{max}}{|\mathbf{p}^{(3)}(u)|_{max}}}, \ldots \right\} \tag{3.7}$$

will satisfy all constraints. However, constant scaling cannot guarantee initial and final velocities to be zero, which is usually a requirement. In this case, some other continuous motion law must be imposed. In case a local interpolation procedure is followed, each segment can be reparametrized separately, instead of considering the same constant scaling λ for the entire trajectory.

A final note on trajectory generation is that it might not be needed for every problem being solved. Many problems often desire the robot to follow a specific path, with no particular regard for the time of motion. In this case, the output of the path planner can be splined and sampled at a desired resolution and a simple point-to-point motion can be employed to follow the path. Details of one such motion controller are given in section 5.3.1.

3.3 Motion control

Motion control determines the time history of the forces/torques to be developed by the robot's actuators in order to guarantee execution of the commanded motion task, while satisfying given transient and steady-state requirements. The solution of a robot motion control problem has been conventionally split into two stages. The first stage is path or trajectory planning where the robot dynamics are not considered. The second stage is trajectory tracking, where the motion is controlled along the predefined trajectory. This conventional split is mainly driven by the difficulties in dealing with complex, coupled manipulator dynamics at the trajectory planning stage. However, the simplicity resulting from this division comes at the cost of efficiently utilizing the robot actuator capacities. More recently, the theory has been developed for optimally controlling robot motion constrained by a pre-defined path. These techniques blur the distinction between trajectory planning and trajectory tracking, but still fall into the category of motion control.

This section begins with a quick overview of control concepts specific to robotics and then focuses on time optimal control along prespecified paths. This is the most relevant type of control in the context of a motion planning and control framework, which has a dedicated path planning component for precalculating geometric paths. The discussion is held in the context of kinematic chain manipulator type of robots, although the principles can also be applied to the motion of mobile robots.

The control scheme for a robot manipulator can be decentralized or centralized. Decentralized control regards the manipulator as formed by N independent systems (the N degrees of freedom, or joints). Each joint axis is considered as a single-input/single-output (SISO) system and the coupling effects between joints due to varying configurations during motion are treated as disturbance inputs. Decentralized control is especially suitable for manipulators with high reduction ratios in their joints. The presence of gears tends to linearize the system dynamics and thus decouple the joints in view of reduction of nonlinearities. This is achieved at the cost of increased joint friction, elasticity and backlash, that might limit the system performance. Centralized control takes into account the dynamic interaction effects between the joints. Individual joint axes are no longer considered as isolated SISO systems. The computation of torque history at a given joint requires knowledge of the time evolution of motion of all joints. Centralized control is used when joints are driven with direct drives and/or high operational speeds. These factors give large nonlinear coupling terms and the configuration dependent inertia, Coriolis and centrifugal forces become significant. Considering them as disturbances induces large tracking errors. Centralized algorithms are designed to take advantage of detailed knowledge of manipulator dynamics in order to compensate for the nonlinear coupling terms of the model. In many cases, the joints are equipped with torque sensors and then these measurements can be conveniently utilized, avoiding online computation of the terms in the dynamic model.

The control of a robot manipulator can be in joint space or in operational (cartesian) space. The control structure of both schemes employs closed loop feedback to improve robustness to modeling errors and reduce the effects of disturbances. In the joint space scheme, the motion requirements in the operational space are translated back into the robot's configuration space (via inverse kinematics) and the controller is designed to track the joint space reference inputs. Quite often, the path/trajectory is specified in joint space and then this is the most natural choice for the control scheme. Joint space control is easier to implement and works directly at the actuator level of the robot. The drawback lies in the fact that the operational space variables tend to get controlled in an open loop fashion, since the reference variables are in joint space, and not in operational space. Thus, any imprecision in the transformation of variables from operational to joint space[7] will cause a loss of accuracy in the operational space variables. Operational space control, on the other hand, operates directly on the operational space variables. The inverse kinematics is then embedded in the feedback control loop. Thus, this is a direct control for the position and orientation of the end effector. In many cases though, this is only a potential advantage, since the end effector pose is not directly measured but calculated from a measurement of the joint space variables via forward kinematics. However, when interaction of the manipulator with its environment is of concern, operational space control often becomes a necessity. For example, when the end effector is in contact with elastic objects in the environment, it is necessary to control the position as well as the contact forces and in such a case, operational space control is more convenient.

The general control theory for robot manipulators is well developed and can be found in any good book on robot control. [48] and [75] are good starting points.

We now turn our attention to the problem of motion control along prespecified paths. The most trivial case is when the path is available as a group of closely spaced points, in which case a point-to-point controller can be used to move from one point to another, under the assumption that the next setpoint is fed when the robot is close enough to the current setpoint. This kind of motion can be manually tuned and a thorough analysis of the manipulator dynamics can be neglected. However, any attempts at finding a time optimal motion along a specified path must consider manipulator dynamics. This is because motion along a path is governed by dynamic equations which are nonlinear. The geometric properties of the path (curvature) will be reflected in different terms in dynamic equations (inertial, centrifugal and Coriolis forces) during the motion.

Similar approaches for optimal control of a manipulator along a given path are presented in [33, 74]. The authors propose a method to convert the limits on the joint torques to limits on the acceleration along the path. By assuming that the acceleration undergoes bang-bang control, a scheme was obtained to identify the switching points. Further, [74] proved that the switching points on the part of the path where acceleration is saturated are finite in number. [69] proposed a method to simplify the computation of the switching points. A method to improve the efficiency of the path-following algorithm is given in [77]. A detailed discussion of how the algorithms have subsequently evolved can be found in [86]. The concept of a path velocity controller in addition to the ordinary robot controller for improvement of path tracking is evaluated in [49].

3.4 Sensing and estimation

The algorithms running in the framework are driven by data. Hence, in order to do anything meaningful, the framework must have access to data at all times. The most important data is that which is obtained through sensors and provides information about the state of the real world the robot is operating in.

In order to be useful, sensor data must be characterized by two properties

1. The data must be timely
2. The data must be accurate

An important interpretation of timeliness is that the most recent data available in the framework must represent the physical signal values at the same point in time. In practice,

[7]The imprecision is usually due to uncertainty in of the robot's mechanical structure (construction tolerance, lack of calibration, gear backlash, elasticity).

there is always some latency in the data acquisition so that the most recent data available indicates signal values at some past time instant. This latency needs to be minimized and algorithm designers should be aware of the presence and consequences of the latency. The latency results from factors internal to the framework as well as external sources. Internal latency can be minimized by careful prioritization and scheduling of the data acquisition code. External sources like bandwidth, network and sensor delays must be analyzed and stochastically modeled.

Stochastic models and estimation are also needed to assure the accuracy of the data. Mathematical models only approximate reality and even then, the data is prone to non-deterministic disturbances. Finally, even sensors do not provide perfect and complete data about a system. Either the desired quantities are not measurable, or they are corrupted due to noise and sensor dynamics.

Accounting for data uncertainties in an algorithm makes it more complex than a similar algorithm which trusts the data it is working with. In order to avoid implementing this additional complexity in algorithms, it makes sense to have a single component in the framework that gathers data and (if necessary) another component that filters and analyzes the data to provide values together with associated degrees of confidence. The rest of the system may then choose to modify its behavior based on the confidence level of the data values.

A lot of estimation theory has foundations in bayesian probability theory. An excellent introduction to bayesian probability theory is given in [40, 38]. The Kalman filter [57] is a popular approach to applied estimation. An extremely good introduction to the concepts of Kalman filtering is given in Chapter 1 of [66]. A practical introduction with examples and results is given in [82]. Theory and approaches to optimal state estimation are given in [76].

Finally, the design of a generic real-time infrastructure for signal acquisition, generation and processing is given in [42].

Chapter 4

Framework requirements

A *framework is a design and an implementation that provides one possible solution in a specific problem domain.*[39] This chapter discusses the important requirements of a generic framework for robot motion planning and control. It presents *what* is needed, without details of *how* these needs should be satisfied[1].

Requirements can be split into design requirements and user requirements.

4.1 Design requirements

Design requirements describe functional objectives of the framework. They define what needs to be done for the framework to achieve its intended function. If a design requirement is not met, the framework will not perform a function at all, or it will not perform it in the desired way.

4.1.1 Must have

These requirements must be satisfied, else the framework will not fulfil its primary purpose.

1. The framework must be capable of executing all the components needed to solve a motion planning and control problem. The capability to execute a component does not necessarily imply the need to do so. Depending on the problem requirements, it should be possible to select the components to be executed. This enables the system integrator to build a customized solution for the problem. Thus, the framework must provide the possibility to run only a subset of all available components. Lack of this possibility would lead to solutions that are an overkill for the selected problem class. It would also impose an unnecessarily high threshold on the minimum computer resources needed to run the framework. Running unnecessary components also increases the complexity of the solution and thereby, the possibility of things going wrong.

2. The framework should be available in the form of a set of services. A combination of these services would be typically used to solve a given motion planning and control problem. For example, the framework can use its functional components to provide services like collision free path planning within a scene, trajectory generation and motion control. These can be invoked one after another by one or more client programs to move a robot among obstacles. A natural solution to this requirement is to have

[1]The *how* part comes in Chapter 5.

at least one server component that accepts and responds to service requests. A conceivable alternative to a server is to have a program that accepts runtime commands and is executed once every time a problem instance is to be solved. This alternative is inferior for a lot of reasons. To begin with, the program must acquire 'situational awareness' of the robot, sensors, workspace, obstacles, user preferences, settings et cetera at every invocation. Knowledge acquired from previous runs cannot be easily maintained to optimize performance. Connecting multiple programs to this program and performing time-bounded, error tolerant data exchange would be non-trivial. Solutions to these and other problems would eventually converge to a "server" paradigm.

3. Robust error handling is an important quality requirement. The framework should not surprise the user by unexpected shutdowns due to errors. Therefore, error handling considerations should be an integral part of the design. This avoids ad-hoc handling of errors during the coding phase.

4. Extensibility is an essential requirement. It is unlikely that every single requirement of the framework will be anticipated in advance. In order to accommodate unforeseen requirements and user demands, it should be easy to extend the framework without making substantial changes to the architecture[2]. The component interfaces evolve over time and this process should not break the interaction with other components.

4.1.2 Good to have

These requirements are not critical in the sense that without them, the framework will still meet its primary goals. However, meeting these requirements will result in a more usable and elegant solution.

1. Communication with the framework server should be independant of the programming language the client is written in, as well as the computer and operating system the client program is executing on. The communication protocol should adhere to a well established and ubiquitous standard. Ensuring this requirement makes it possible for the framework to serve the broadest possible range of clients. It imposes minimum constraints on how the clients should be written.

2. The framework should be network aware, so that clients need not be running on the same physical computer as the framework. Network awareness also provides the possibility of teleoperation of the robot.

3. Communication with the server component should be non-blocking. Non-blocking means that a client should not have to issue a service request and wait indefinitely for a response from the server. Rather, the server should acknowledge the client request as soon as it is received. The client may then send status requests to determine whether previous requests were successfully completed.

4. A functional component within the framework must have the ability to execute different algorithms that provide the same functionality. This is because the algorithms needed for each function differ depending on the problem being solved. For example, a motion control algorithm for an arc welding robot will differ from that of a pick and place robot. It should be possible to change the algorithm to be executed within each component, without recompiling the framework. Such a structure ensures that

[2]"With proper design, features come cheaply." - Dennis Ritchie

the framework is not restricted to a predefined set of algorithms. The system integrator can select or write an algorithm that suits his application the best. The desired algorithm may be specified at system startup. Also good to have is the ability to switch during runtime between algorithms which were available at system start. The ultimate in flexibility would be the possibility to write a completely new algorithm and run it without restarting the system.

5. Each component of the framework should have a well-defined interface, inputs and outputs, while the working of the component should be a blackbox. An important implication of this is that it is not necessary to come up with a comprehensive, uniform representation for scene files, robot representations, obstacle definitions and so on. A component can choose to define its own representations of the objects it works with, independantly of and opaque to other components. This saves a significant amount of work for the framework designer. Also, the components can then use third party libraries with formats native to the particular library being used. Essentially, each component should be complete in itself.

6. The functional components should be as loosely coupled as possible. Loose coupling means that a component does not need to have knowledge about the identity, service interfaces, locations or internal states of other components. For example, the communcation between components is decoupled if the "sender" does not have to know the identity of the "receiver" component[39]. Loose coupling makes it very easy to make modifications to the framework architecture and to extend it in ways unforeseen during the first design iteration.

4.1.3 Wish list

1. The framework itself shouldn't necessitate the use of a particular operating system or hardware architecture. It is okay to specify minimum hardware resource requirements in order to run the framework.

2. It should be possible to distribute execution of the framework components across different physical computers. This way, a resource intensive component can get its own hardware and won't hinder the performance of other components.

3. The framework should offer the possibility to execute components in real-time. Algorithms which assume deterministic scheduling are simpler to design. Also, real-time can be a non-negotiable requirement[3] for the working of a component. For example, a motion controller might not be able to achieve the desired performance in the presence of scheduling jitter.

4. A reporter component should be present to stream status and data content from the framework to interested client programs. This way, users can write programs that 'listen in' to what is going on inside the framework and extract the particular data of interest. An example is a visualizer program that can graphically represent the robot in its workspace, display planned paths and track the robot motion as it happens.

[3]The reason this requirement is not a 'Must have' is because code that needs realtime can be run on separate processors with lower latency. It is more convenient, however, to run it within the framework itself.

4.2 User requirements

User requirements describe what needs to be done to provide a good usage experience of the framework. The framework can meet every functional requirement and still be difficult to use. Defining and satisfying proper user requirements ensures that this is not so.

1. The framework should be as easy to use as possible. Complexity is the price for power, but it should not be necessary for the user to thoroughly understand every detail of the framework implementation before using it. The learning curve should be gentle.
2. The framework must accept commands both in cartesian and configuration space.
3. At all times, the user must be able to examine the status of various components in the framework to know exactly what is going on.
4. Parameters that affect performance of a component must be changeable while the component is running. This makes performance tuning easier, faster and more convenient.
5. Settings must be remembered. Thus, when a parameter is tuned to a correct value, the value must be stored and used the next time the framework is started.
6. Programming libraries should be provided so that using the framework via client programs is matter of calling the appropriate library function.
7. A visualizer client must be available with the framework so that the user can graphically inspect the output of a planning or motion process.
8. Usage and failure modes must be well documented so that the user knows exactly what to expect. The framework must adhere to the rule of least surprise [71].
9. Errors must be reported verbosely with details of the fault and (if possible) hints on how to rectify it.
10. Good documentation must be available so that the user can form a coherent mental picture of the framework.

Chapter 5

Implementation

This chapter presents the framework as it is actually developed. It begins with a discussion of the overall framework structure and then provides details of some individual components.

5.1 The framework structure

The framework is written in the C++ programming language[1]. This language was chosen because of the wealth of available code and tools that could be directly used in the project. Concerns regarding C++ overheads and performance are considered in [84], which presents techniques for effective use of C++ and demonstrates that with a careful selection of language features, current C++ implementations can match hand-written low-level code for equivalent tasks.

The framework is implemented as a set of simultaneously running components, shown in figure 5.1.

This architecture was inspired by [43]. Orocos [41, 37] is used to implement the framework structure. Orocos is a freely available, general purpose, modular, cross-platform software which can be used for building frameworks intended for robot and machine control. Features which made Orocos a compelling choice for building the framework were

1. Ready availability of the "software component" pattern[2]
2. Rich set of facilities available for safe and reliable inter-component communication
3. Possibility of scheduling components as realtime/non-realtime with differing priorities
4. Provision of an OS abstraction layer which minimizes operating system specific programming code
5. Possibility of distributing components seamlessly across multiple computers
6. Excellent documentation and active community

Each component can independently execute an algorithm or state machine relevant to the task it is supposed to perform. Interaction among components takes place via component interfaces. A component interface is essentially a set of functions. The interface can be

[1]Strictly speaking, ISO/IEC 14882:2003 Programming Language C++

[2]A software pattern describes a proven way to solve a commonly occuring problem. Here, Orocos provides a way to implement the software component pattern.

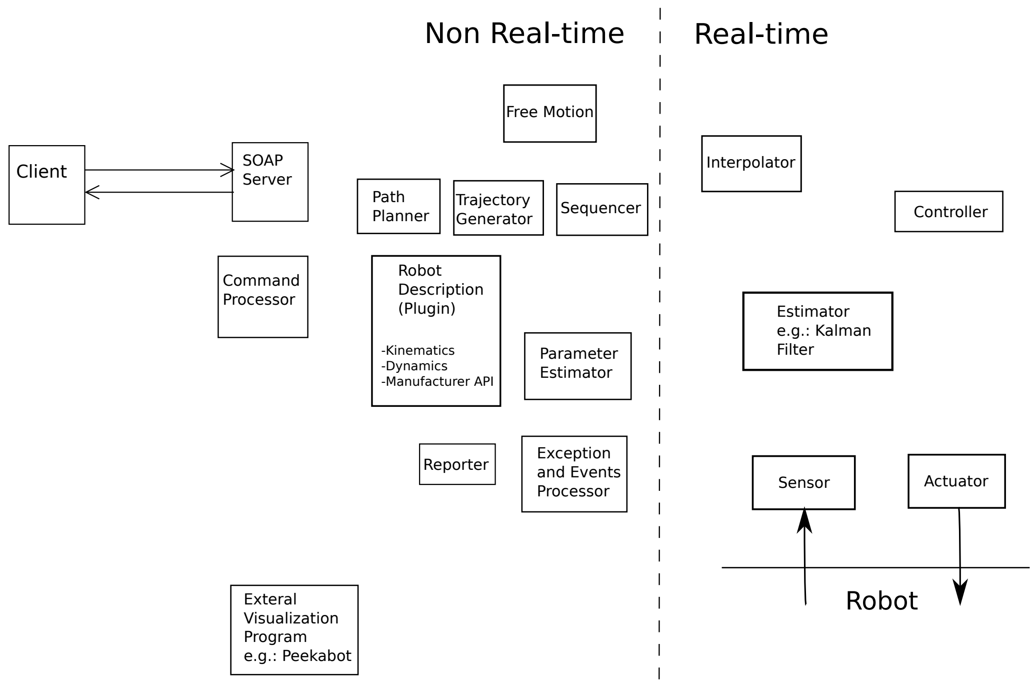


Figure 5.1. The Sarathi framework

extended to accommodate more functions as needed. The intention of the current implementation is not to define exhaustive interfaces, but to have something that works and is extensible for future needs, while retaining backward compatibility.

In Orocos terminology, a component is referred to as a 'TaskContext'. The terms 'component' and 'TaskContext' will be used interchangeably in this report. The functions of each TaskContext in figure 5.1 will now be discussed before delving deeper into the anatomy of a TaskContext.

5.1.1 The role of each component

- SOAP Server (Aperiodic): A TaskContext responsible for managing communication with clients using SOAP messages. It also forwards command requests to the Command Processor. More details can be found in section 5.2.
- Command Processor (Periodic): A TaskContext which executes commands received from the SOAP server (and optionally from other task contexts)
- Path Planner (Aperiodic): Upon receiving a goal position, the planner plans a path from the current position to goal position. The result of a successful planning run is a set of key configurations which must be successively attained in order to reach the goal in a collision free manner. The path planner needs a scene description, which describes the robot and the obstacles in its environment
- Trajectory Generator (Aperiodic): The path planner outputs a set of points, without any consideration of the time at which each point is to be reached. The trajectory generator imposes timing constraints which define the time at which each point must

be reached. The output of the trajectory generator is a set of points, where each point is associated with a time when it should be reached.

- Sequencer (Periodic): The Sequencer evaluates whether the generated trajectory is being executed as expected, and makes corrections if it is not so. The evaluation involves evaluation of application specific conditions and uses the output of the Trajectory Generator and the realtime and non-realtime estimators
- Interpolator (Periodic): Generates a motion set-point every time it executes. Could use different methods like cubic splines, trapezoidal profiling etc.
- Sensor Value Estimator (Periodic): Could be a Kalman filter or Luenberger observer. Generates filtered sensor readings for variables like robot position, velocity, acceleration etc. The output of the estimator is a sensor reading with an estimated confidence level, which can be used by the other components.
- Parameter Estimator (Periodic): Used to estimate parameters of models that other components rely on. Example could be identification of dynamic system load
- Exception and Events Processor (Periodic): Used for fault management and a catch-all for unhandled events in the system
- Free Motion (Aperiodic): Permits a total bypass of planning (and possibly motion control), in case it is necessary. This is for raw, low level joint motions or for direct invocations for robot manufacturer API
- Reporter (Periodic): The Reporter gathers data from other components in the system and outputs it in a continuous stream. External programs can subscribe to this stream to know the internal state of the system. An example could be a visualization program that shows the planned path, current robot configuration and so on
- Robot Description (Aperiodic): Each robot being used needs to be characterized by kinodynamic constraints and several other attributes. Robot specific data enters the system through this component. Among other things, it provides links to functions in the robot manufacturer's API, which can be invoked for actual robot motion.

5.1.2 Anatomy of a component

"A component is a basic unit of functionality which executes one or more (real-time) programs in a single thread. The program can vary from a mere C function over a real-time program script to a real-time hierarchical state machine. The focus is completely on thread-safe time determinism. Meaning, that the system is free of priority-inversions, and all operations are lock-free (also data sharing and other forms of communication such as events and commands). Real-time components can communicate with non real-time components (and vice verse) transparently." [79]

The following description of a component is mostly verbatim from various parts of [79]. A component can be interfaced in five distinct ways

1. Data Flow Ports: are thread safe data transport mechanisms to communicate buffered and unbuffered data between components.
2. Properties: are run-time modifiable parameters, stored in XML files. For example: "KinematicAlgorithm", "ControlParameter", "HomingPosition"

3. Methods: are callable by other components to 'calculate' a result immediately, just like a 'C' function. They are run in the thread of the calling component. For example: "getTrackingError()", "openGripper()", "writeData("filename")", "isMoving()"

4. Commands: are 'sent' by other components to instruct the receiver to 'reach a goal'. For example: "moveTo(pos, velocity)", "home()",... A command is executed in the thread of the called component and therefore cannot, in general, be executed instantaneously[3], since the calling component is using the processor at the time of invocation. Hence the caller should not block and wait for its completion. But the Command object offers all functionalities to let the caller know about the progress in the execution of the command.

5. Events: allows functions to be executed when a change in the system occurs. For example: "Position Reached", "Emergency Stop", "Object Grasped"

Besides defining the above component communication mechanisms, Orocos allows writing hierarchical state machines which use these primitives. This is the Orocos way of defining application specific logic. State machines can be (un-)loaded at run-time in any component.

The schematic interface of a component is represented in figures 5.2 and 5.3.

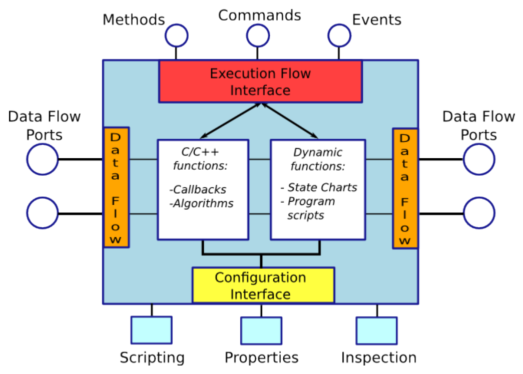


Figure 5.2. Component schematic

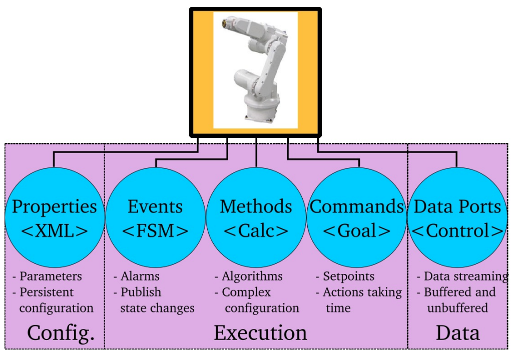


Figure 5.3. Component interface

The task specific code of a component is written in 'Hook' functions. A hook function is called whenever a specific function in the component API is invoked. Some hooks are called whenever a component changes its state. The runtime states of a component are shown in figure 5.4, the error states are shown in 5.5.

In its simplest possible form, a periodic component executes in the following 3 steps

1. the startHook() function is called once whenever the component first starts executing

2. the updateHook() function is called at each successive period

3. the stopHook() function is called once just before the component stops executing

5.1.3 Intercomponent communication

Communication between components occurs through the five interfacing facilities described in section 5.1.2. Components need to be connected before they can use each other's interface. The connection can be uni- or bi-directional. In a uni-directional connection, only one

[3]On a uniprocessor system.

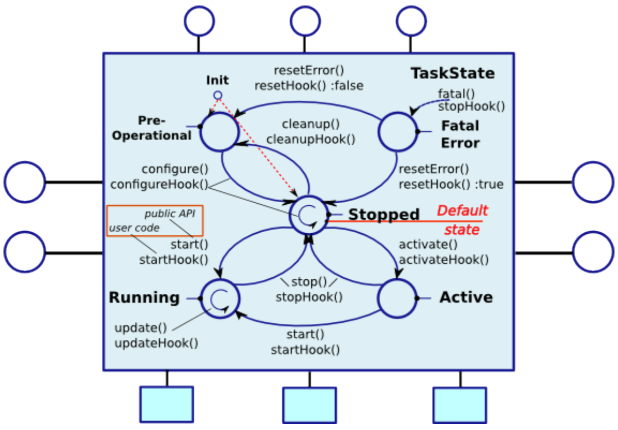


Figure 5.4. Component runtime states

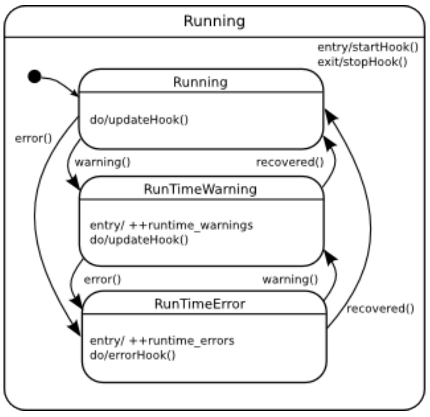


Figure 5.5. Component error states

component can use the interface of the other, while in a bi-directional connection, both components can use each other's interfaces. See figure 5.6. This allows to build strictly hierarchical topological networks as well as complete flat or circular networks or any kind of mixed network.

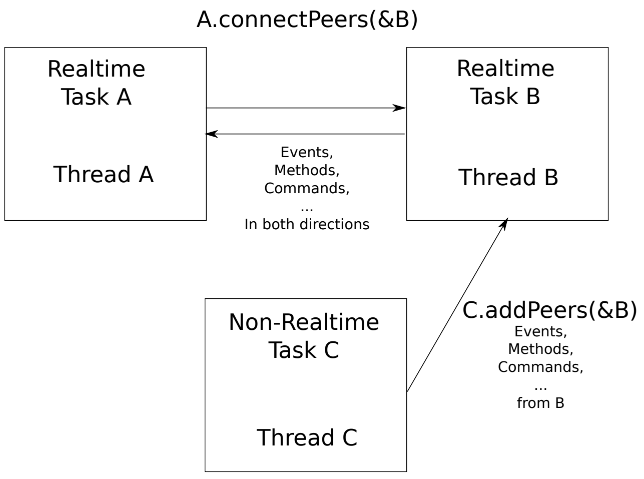


Figure 5.6. Connecting components[79]

Connected components are called 'Peers' since there is no fixed hierarchy. The peer connection graph can be traversed at arbitrary depth i.e. a component can access its peer's peers.

Components can exchange data using the dataflow ports. The direction of data flow is imposed by the read/write nature of the ports. Figure 5.7 shows some possible topologies. Four components, "A", "B", "C" and "D" have each a port "MyData" and a port "MyData2". The example demonstrates that connections are always made from writer (sender) to reader (receiver).

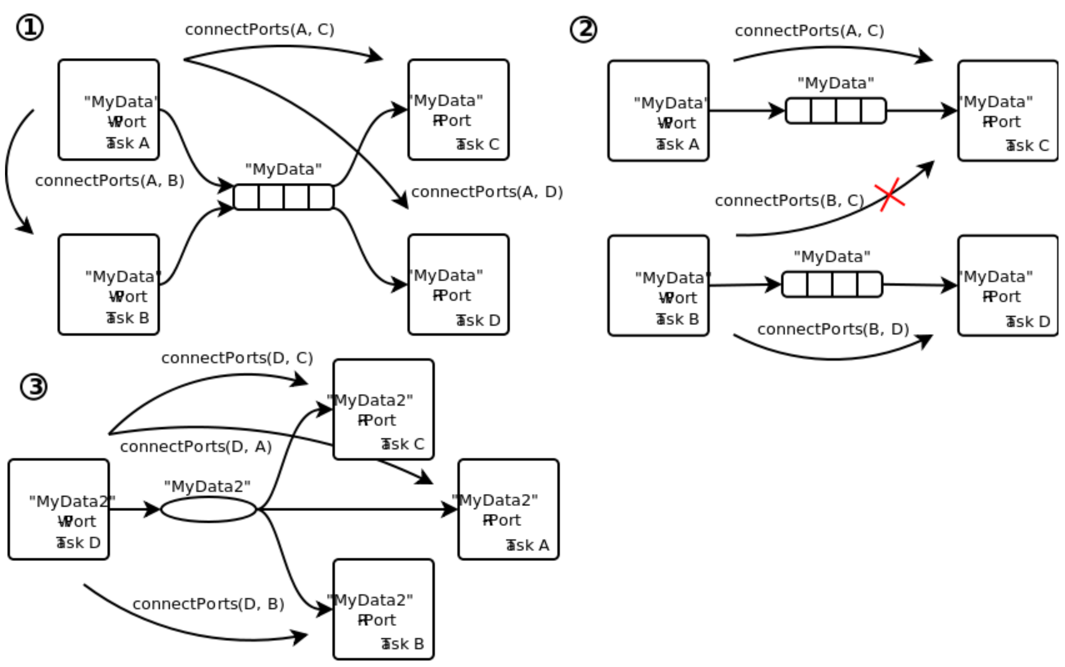


Figure 5.7. Example dataflow networks[79]

5.1.4 Plugin management

The functionality of a component is provided by the algorithm or state machine it executes. There can be different algorithms (or state machines) for the same task, depending on the specific problem being solved. For example, the motion controller component would use a different algorithm if a pick-and-place type of problem is replaced with an arc-welding problem.

The use of plugins separates the *interface* from *engine* and allows a component to execute a user selected algorithm at runtime. Plugins have been implemented using the Qt plugin management framework [6].

When a component starts, the following sequence is executed

1. Read configuration file to know which plugin is to be loaded
2. Search standard locations for desired plugin
3. If plugin is found and successfully loaded, enter running state, else enter error states

Once a plugin is successfully loaded, the entire functionality of the component is handled by the plugin. This is done by making sure that every component interface function merely calls the equivalent function in the plugin.

Plugins are good because they enable system integrators to extend the framework in extremely specific ways, without touching the framework code.

5.2 The SOAP server component

The SOAP server component provides the interface via which client programs access the services provided by the framework. This is accomplished by using the gSOAP [24] library which implements the SOAP [35] protocol.

SOAP is a simple XML-based protocol to let applications exchange information over HTTP. The protocol is platform and programming language independent and thus there are no restrictions on how the client programs need to be coded. Network awareness is inherited from the HTTP protocol. Another protocol which meets the requirements is XML RPC [23]. However, SOAP was chosen over XML-RPC because of the availability of a good library implementation in the form of gSOAP. CORBA[3] and ICE[7] are two additional methods to invoke functions and exchange data across computers, but they were not considered because they exchange binary data and need a mapping to the programming language being used. SOAP, on the other hand, uses plain text and hence no language specific mappings are needed.

"The gSOAP toolkit is an open source C and C++ software development toolkit for SOAP/WSDL and XML Web services and C/C++ XML data binding applications that benefit from automatic XML serialization." [5]. In simpler terms, gSOAP permits a client program to call a C++ function which then automatically calls an equivalent C++ function in a server program. The server program can be running on another computer and gSOAP uses the SOAP protocol for communicating the function arguments and response data over the network. gSOAP also makes it possible to create a server object which can listen on a specific port number and act on received requests. Thus, the entire protocol and communication effort is handled transparently and the developer becomes free to concentrate on *what* should be communicated, instead of *how* it should be communicated.

When the framework starts, the SOAP server component is started and it starts listening on a specific port for incoming requests. When a request is received, the server either fulfills the request immediately or dispatches it to the Command Processor component for subsequent execution. A request is executed immediately if it can be serviced in a deterministic and short time interval. For example, a request for data already available in other parts of the framework is serviced immediately. Examples include requests for the planned path, the current robot position, the status of the various components et cetera. On the other hand, requests that could take a large or indeterminate amount of time are dispatched to the Command Processor component where they can be scheduled for execution in separate threads. Examples include requests for planning a path, moving the robot to a specific position et cetera. This model of servicing requests ensures that the clients always get an immediate response either in form of real response data or an acknowledgment of their request. Thus, the non-blocking communications requirement is satisfied.

The networking code in the component is handled using the cross-platform Qt Network module [18]. This ensures platform independence of the networking code.

5.3 The motion control component

The motion control component is responsible for moving the robot along the planned path. The motion control algorithm is implemented in a plugin which is loaded when the component is started. The component interface is shown in table 5.1.

Since the motion controller moves the robot along a specific path, the interface contains functions to set and erase this path. Additionally, the motion controller can use the preferredPathSpace attribute to provide a hint about whether it prefers a path specified in configuration or cartesian space. The positionTolerance attribute specifies the tolerance within which the desired end position is to be achieved. The exact interpretation of its value is dependent on the motion control algorithm being used.

Commands	Methods	Properties/Attributes
execPath	setPath	status
erasePath	stop	pathSpace
init		preferredPathSpace
reset		positionTolerance

Table 5.1. Motion Control Component interface

The motion control component is initialized when the framework starts. For all subsequent motions, the planned path is transferred to the component using the setPath method. Once the path is properly set, the execPath command is invoked in order to move the robot. The interface functions are simple wrappers around relevant plugin functions and there are no hard and fast rules defining exactly what happens when an interface function is called. The plugin author has total freedom to define the system behavior.

5.3.1 The time-invariant motion controller

At the time of this writing, exactly one motion control plugin has been implemented, which will be documented in this subsection.

Background

The researchers at KTH/CVAP needed to move a KUKA KR5 Sixx R850 industrial robot arm [8] during their experiments. Accuracy of positioning was more important than the velocity. Also, the exact path between point to point motion was unimportant as long as there was no collision with the obstacles in the environment. The manufacturer provided an API function to move each joint to a desired position, at a desired velocity. Trapezoidal velocity profiling was employed between point-to-point motion. The path followed while moving from one point to another was determined by the proprietary robot motion controller and not controllable by the user. The robot would be sent motion commands over a non-dedicated network. Due to this and the proprietary processing involved in the robot controller, there would be stochastic delays in the execution of commands.

Under these operating conditions, the requirement was for a motion controller that could guide the robot's end effector along a specified path in cartesian space.

Solution

The problem constrains the path of the robot, not its trajectory. Thus, there is no requirement of *when* each point on the path is to be attained. Since timing constraints are not imposed, the motion controller is made time invariant. This means that the value of the control signal does not depend directly on time.

The path to be followed is an ordered set of points. These points are usually generated by the path planner and no assumptions can be made about the distance distribution between the points. The motion controller fits a cubic spline[54] through the points in order to get a C^2 curve that passes through every point in the path. Then, points are uniformly sampled along this spline and used as successive set-points for the robot motion. A point on the path is generally multi-dimensional and the program GNU Spline, which is part of the plotutils package [4] is used to spline the path.

Once the path is splined, algorithm 1 is periodically invoked for motion control.

Algorithm 1 The time invariant motion controller

```
 1: Get current position
 2: if (Moving for first time) then
 3:     Assert that gripper has not moved
 4:     goto next point on splined path
 5: end if
 6: Assert robot is not stuck in same position
 7: if (At last point on splined path) then
 8:     Exit from current iteration
 9: end if
10: if (Close enough to point being approached) then
11:     Assert that gripper has not moved
12:     goto next point on splined path
13: end if
```

Thus, the motion controller moves the robot along the points on the splined path. The number of points on the path can be controlled, to achieve a sufficient point density so that the resulting motion is collision-free and smooth. The accuracy of the motion i.e. how closely the robot tracks the desired path can be tuned by defining how close the robot should get to a point before it is given the next set-point. This "close enough" distance can be a dynamically calculated quantity based on the current robot velocity.

If there is a requirement for constant velocity, the theoretically latest position at which the next set-point should be fed is shown in figure 5.8. Since the robot manufacturer's controller performs a trapezoidal velocity profiling, the next set-point should be provided before the robot starts decelerating. The velocity and point density is tuned such that the robot is close enough to the targeted set-point by the time the theoretically latest position is reached. This effectively gives a trapezoidal velocity profile over the entire path being followed.

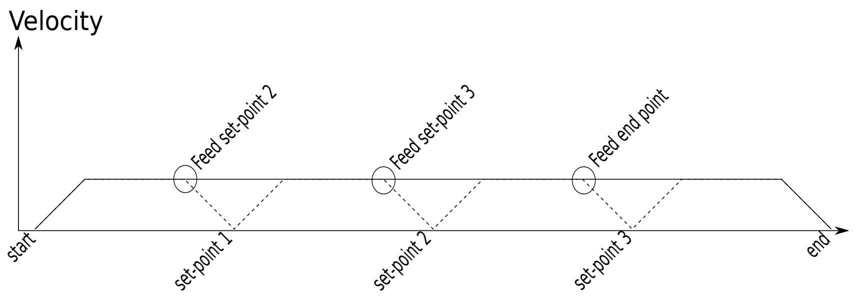


Figure 5.8. Specifying set-points

5.4 The path planner component

The path planner component is responsible for generating a collision free motion path for the robot. The input to the planner is the geometry of the robot and obstacles together with an ordered set of positions which the robot must reach while in motion. Generally, these positions are the start and end points of the motion, but can sometimes include specific positions needed to be reached along the way to the end point. The planner then generates a path that connects these positions with collision free segments.

The interface of the planner component is shown in table 5.2.

Commands	Methods	Properties/Attributes
planPath	setKeyPoints	plannerStatus
	getPath	planningStatus
	setPlannerFile	plannerFile
	setSpaceType	spaceType

Table 5.2. Planner Component interface

The setKeyPoints method is used to tell the planner the positions along which planning is to be done. getPath returns the planned path. setPlannerFile passes a file to the planner which describes the environment the robot is operating in. The environment consists of the robot, the obstacles and their geometries in space. setSpaceType is used to define whether the key points provided are in configuration or cartesian space. Once the methods are invoked to describe the planning problem, the planPath command can be used to actually solve the problem. The planningStatus attribute indicates the result of the planning process.

The actual work of planning is done by a planner plugin which is loaded by the component at startup. All the component interface functions call the relevant plugin methods.

5.4.1 The MPK planner plugin

At the time of this writing, exactly one path planning plugin has been implemented, which will be documented in this subsection.

How the planner was chosen

There is a fair amount of code already written for motion planning. When choosing a planning algorithm, the top two concerns were

1. The ease with which the planner could handle the problem of planning for the robots we intend to use

2. The availability of the algorithm in the form of a ready to use library or source code

The first application of this framework would be for moving industrial robotic arms at KTH/CVAP. These are fixed base kinematic chains. The algorithm should be able to work with these types of robots. Next, it is easier to integrate the planner into the framework if it is available as a well-documented software library. This saves the effort of reading a reference paper and coding the algorithm from scratch. With this in mind, a survey of available software was made and some potential candidates were identified.

The Motion Strategy Library (MSL)[12] provides a number of planning algorithms. However, it is dated with respect to the versions of software it is written in. The last release was made in 2003 for Redhat Linux 8.0. It was difficult to even compile it on a recent distribution of Linux. Also, it doesn't work on Mac, thus compromising on the cross-platform requirement.

The Motion Planning Kernel (MPK)[10] from Simon-Fraser university is another framework for motion planning algorithms. However, it works only on Microsoft windows, compromising the cross-platform goal.

OOPSMP[13] from the Kavraki Lab at Rice university is a framework for research on motion planning algorithms. It is directed more towards mobile robotics and at the time of evaluation did not support an easy way to work with robotic arms.

OpenRAVE[15] provides a number of path planners and works well with robotic arms. However, OpenRAVE is much more than mere motion planning and considerable other software infrastructure would be needed in order to use the OpenRAVE planners.

The Motion Planning Kit (MPK)[11] from Stanford is a C++ motion planning library with minimal software dependencies. It works on Microsoft Windows, Linux and MacOS.

Ultimately, the Motion Planning Kit was chosen for the following pragmatic reasons

1. It was capable of solving motion planning problems for one or more industrial robot arms
2. Defining robot models and obstacles was extremely easy
3. It understood the concept of gripper degrees of freedom and ignored them for motion planning
4. It came with well-written examples and very good facilities for visualization of generated paths
5. The documentation was excellent. The licensing was open.
6. The author is extremely responsive and readily replied to all queries

The next section describes the MPK planner in more detail and how it is used within the planner component.

The MPK algorithm

The Motion Planning Kit (MPK) is a C++ library and toolkit for motion planning for one or more robots. It is a Single Query Bi-directional Probabilistic Roadmap Planner with Lazy Collision Checking (SBL). Details of the SBL algorithm can be found in [72]. MPK uses a fast dynamic collision checker that guarantees not to miss any collisions. Details of the collision checking algorithm can be found in [73].

MPK works in configuration space. Robots and obstacles can be defined and added to the planner without recompiling the planner code. Both the robot as well as the obstacle geometries are defined as solid models in the Open Inventor[14] format. More information about working with Open Inventor can be found in [83]. The complete description of the robots and obstacles (as against a description of just their geometries) is in a MPK specific format derived from the Open Inventor format. Defining new robot models is described in [28], while [29] describes how new scenes are created.

MPK accepts any number of configuration space positions of the robot(s) and plans a collision free path connecting them in the given scene. The output of the algorithm is an ordered set of points in the robot's configuration space which describes the generated path. The generated path is based on randomly sampled points in space, and so the path may need to be smoothed. For this purpose, MPK includes a path smoothing function.

In case the user chooses to specify the robot positions in cartesian space, an inverse kinematics solver needs to be called in order to convert the cartesian point to configuration space.

5.5 The robot component

The framework needs a consistent interface to the robot being controlled. This consistency is made possible by the robot component which provides a uniform interface for use by the other system components. Thus, the task of the robot component is to abstract away the differences between different robots controlled by the framework. The component does this by using plugins. The component-plugin interface is uniform and well-defined, whereas the plugin-robot interface can be highly tuned to the robot being controlled.

The robot component interface is shown in table 5.3.

Commands	Methods	Properties/Attributes
gotoPosition	stop	robotDOF
	getPosition	gripperDOF
	ikSolver	status
	fkSolver	hostName
		hostPort

Table 5.3. Robot component interface

The gotoPosition command can be used to move the robot to a specified position, while the getPosition command returns the current robot position. Both these functions work with values in configuration as well as cartesian space. the ikSolver and fkSolver functions are used for solving inverse and forward kinematics problems respectively. robotDOF and gripperDOF indicate the degrees of freedom in the robot and the gripper respectively. These values may be needed by other components for various reasons. hostName and hostPort are used to hold the host name and port number of the server controlling the robot is running on. Typically, this is the software provided by the robot manufacturer.

5.5.1 The KUKA KR5 Sixx R850 robot plugin

This plugin was developed for controlling the KUKA KR 5 sixx R850 robotic arm [8] at KTH/CVAP vision laboratory. The robot arm has a Schunk dexterous hand [21] attachment for grasping tasks. The robot is used for grasping objects kept on a table. It is required that the robot must not knock over other objects when reaching for the desired object.

The hand is a 7 degree of freedom system (DOF), while the robot arm has 6 DOF. The plugin views the arm-with-hand as a 13 degree of freedom kinematic chain. The first 3 degrees of freedom are used for positioning in cartesian space, the next 3 degrees of freedom are used for orientation in cartesian space. The last 7 DOF are considered passive and are not used for path planning purposes. This is in order to prevent generating a path where the finger joints have to be moved in order to get around an obstacle. Moving the finger joints will likely cause the hand to drop the object it has grasped and is not desirable. Although the last 7 DOF are not changed for path planning, knowing their values is necessary in order to know the pose of the hand at the instant when planning is to be done. Knowledge of the hand pose is necessary to avoid collision with obstacles in that particular pose.

The robot and the hand are controlled via two separate servers which abstract the exact mechanisms used to control them. The plugin therefore interacts with these servers in order to perform operations on the robot/hand. The servers can execute on separate computers and the plugin communicates with them over the network. Communication takes place over the User Datagram Protocol (UDP) instead of the more common Transport Control Protocol (TCP). The reason is that UDP is a connectionless protocol without the overhead of detecting and re-transmitting lost packets. The packets can be self-contained and so only the latest packet received is relevant. Experiments and further details regarding the advantages of UDP over TCP are given in [67] and [78].

5.6 The libhyperpoint library

The components of the framework are complete in themselves. Each component can have its own representation of the data it works with. However, when it comes to data exchange between the components, or with clients, it helps to define standard data types for exchanging the most commonly used data. In absence of a standard data type, components exchanging data would have to know each others data types. Therefore, the concept of loose coupling among components would be lost.

The most commonly exchanged data between the framework components and between the framework and clients is a point or a set of points. In this context, a point can be in n-dimensional real space, R^n. For example, the configuration space of an n DOF robot is in R^n, while a point in cartesian space is in R^6. Therefore, a data type is needed to hold an arbitrary dimensioned point. Let us denote an arbitrary dimensioned point as a hyperpoint. To denote a set of hyperpoints a data type is needed to hold an arbitrary dimensioned array of hyperpoints.

Hence, a library (called libhyperpoint) was created which defines the hyperpoint and hyperarray classes. Each class has methods which facilitate operations on the data type. The API documentation of libhyperpoint can be found at [9].

5.7 The libsarathi client library

Libsarathi is a C++ library which can be used for communication with the Sarathi framework. The library defines a serverProxy class which can be instantiated by client programs. Methods of this class can then be invoked in order to use the services offered by the framework. The library uses gSOAP to handle the communication with the server. Additionally, it needs the supporting library, libhyperpoint, in order to use the hyperpoint and hyperarray data types.

The library functions throw exceptions when they detect error conditions. Errors reported can be communication errors as well as errors in the framework when executing client requests. The exception object contains information about the cause of the error as well as hints on correcting the error.

The library API documentation is described in [30], while [31] provides a usage guide and examples of how to use the library.

Chapter 6

Testing

After a software is designed, it must be tested to verify that it meets the design requirements. This chapter presents how the framework was tested.

The framework exists as a set of services and testing the framework involves testing of each individual service, as well as how the services interact with each other to solve a problem. The services are designed to be driven by client programs and hence, client programs have to be written to emulate real-world use cases. Practically, the functionalities of the framework and the clients evolve simultaneously. As soon as a function is added to the framework, the client is updated so that the new function can be tested.

This chapter now discusses the test tools that were written, followed by the testing methodology employed. Finally, a real use case is presented to show one way the framework is used in reality.

6.1 Test tools

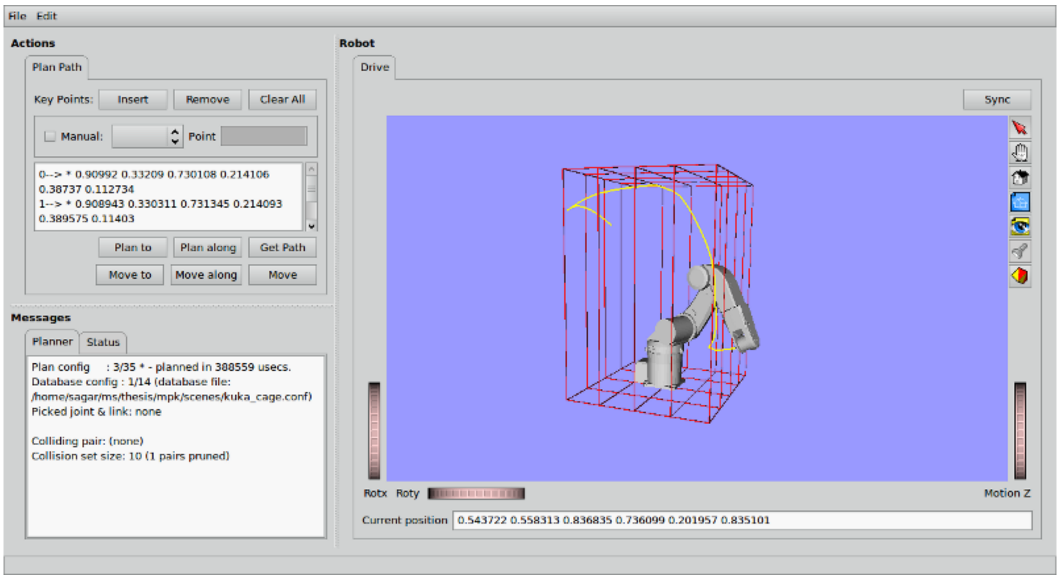
Two client programs were written in order to interact with the framework. These client programs are useful for three main reasons

1. The clients generate meaningful testing data
2. They can be used for systematic testing of each service offered by the framework
3. They are good examples of using the framework and the libsarathi library (for future users)

The client programs are named 'Parth' and 'APIease' (pronounced: appease). Parth is useful for visualizing scenes and generating test data. APIease is useful for thorough testing of framework services. Both clients use the libsarathi library.

6.1.1 Parth

Parth is a software client that can display the scene file used by the MPK path planner. It displays a 3D view of the robot as well as its operating environment, including the obstacles. In addition to display, Parth also allows the user to interact with the virtual robot on the screen. The user can move the robot to any position in the workspace and collisions with obstacles, if any, are immediately shown. The current position of the robot is continuously displayed. Therefore, the user can readily obtain coordinates of valid robot positions in the workspace. This is useful to have because a large number of robot positions are needed for testing the framework. Figure 6.1 shows a screenshot of Parth with the KUKA KR 5 Sixx R850 robot inside a cage obstacle.



```
0--> * 0.90992 0.33209 0.730108 0.214106
0.38737 0.112734
1--> * 0.908943 0.330311 0.731345 0.214093
0.389575 0.11403
```

```
Plan config     : 3/35 * - planned in 388559 usecs.
Database config : 1/14 (database file:
/home/sagar/ms/thesis/mpk/scenes/kuka_cage.conf)
Picked joint & link: none

Colliding pair: (none)
Collision set size: 10 (1 pairs pruned)
```

Current position: 0.543722 0.558313 0.836835 0.736099 0.201957 0.835101

Figure 6.1. Parth screenshot

Parth can also store user-chosen positions and send them to the framework as keypoints along which the path should be planned. The path planned by the framework can be retrieved and displayed in the 3D view. Further, the robot model can be animated to show how the robot would follow the retrieved path. Thus, the user can do a complete simulation of path planning and motion before the physical robot is moved. Parth can also send motion commands to the robot via the framework, and synchronize the current position of the robot model to the actual position of the real robot.

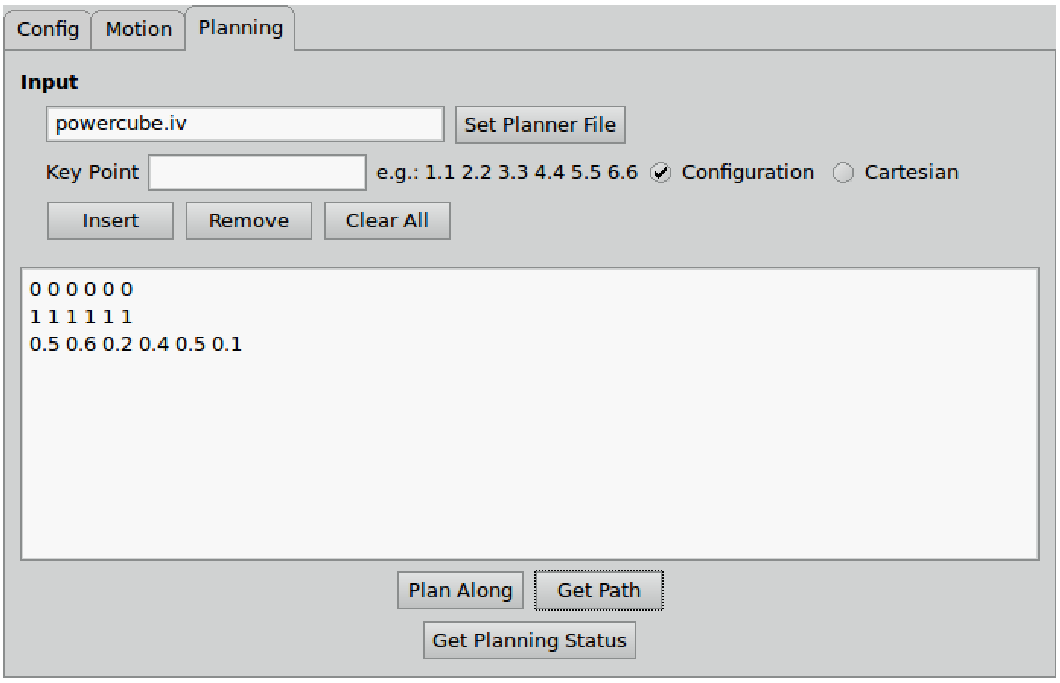
Parth therefore becomes a complete test bed for tasks like testing new robot models, testing new scenes with a variety of obstacles, planning paths among obstacles, moving the robot along planned paths and so on. It can also be used as a GUI to observe the framework functioning while another client program is using the framework services. The 3D visual representation makes it intuitive and obvious to understand what is going on.

6.1.2 APIease

APIease is a software client geared towards testing each function of the interface which the framework uses to communicate with clients. If a function exists in the interface, APIease includes the facility to call it. A screenshot of APIease is shown in figure 6.2.

Clients like Parth are useful for solving specific problems, not for stress testing the framework interface. This is because they sanitize their inputs, call the correct functions with their proper arguments and in general, follow all the rules of correctly operating the interface. They do not deliberately try to break the framework's way of operation by giving bad inputs.

APIease can be used to test the robustness of the framework functions and client communication. This is because it provides facilities to make raw function calls and examine the results. So it is very easy to provide unexpected inputs. With APIease one can



powercube.iv

[✓ Configuration] Cartesian

```
0 0 0 0 0 0
1 1 1 1 1 1
0.5 0.6 0.2 0.4 0.5 0.1
```

Figure 6.2. APIease screenshot

1. Call a function with bad or invalid arguments and observe what happens
2. Call functions in improper sequence. For example, send a Move command followed by a Plan Path command
3. Call random functions while the framework is busy processing previous requests
4. View intermediate results in a function sequence, since each function is called step-by-step
5. Use invalid communication parameters or simulate communication faults like timeouts and delays
6. Call functions when their assumed pre-conditions are false. For example, calling a plan path command without setting the scene file or sending a Move command without first planning a path

Ultimately any interaction of a client with the framework can be broken down into a sequence of function calls together with specific arguments. This sequence can be executed using APIease and so APIease can be used for simulating test cases with any client. Similarly, conditions from bug reports can be duplicated, by sending the function(argument) sequences that cause the bug. This aids in tracking down and fixing bugs.

APIease is therefore a valuable tool not just for the framework developers, but also for users trying out new ideas.

6.2 Testing methodology

Typical use of the framework consists of moving the robot along a specified set of positions, in the presence of obstacles. The process of doing this can be divided into a set of discrete operations. These operations are

1. Initiating communication with the framework
2. Describing the scene to the path planner (scene includes description of the robot, obstacles, their geometries and relative positions)
3. Setting the key positions along which path planning is to be done
4. Planning the path
5. Moving the robot along the planned path

The testing methodology employed primarily focuses on these discrete operations, in the order in which they are executed.

SOAP server testing consists of verifying three aspects: Network capabilities, data transmission fidelity and error reporting. gSOAP permits the creation of a TCP/IP network server, which must be tested with connections from the local machine as well as network machines. The behavior must be defined for cases of dropped connections, simultaneous connections, corrupted data packets, latencies in communication and so on. Data transmission fidelity involves transferring all the data types used by the framework and in the framework-client exchange in order to verify that there is no loss of precision during communication. Finally, networking errors as well as errors in the framework functioning must be reported back to the clients, with additional error data wherever possible. gSOAP provides a built-in mechanism to report these errors and hence a custom error reporting scheme was not necessary. However, testing of this feature involves generating errors on the server side and assuring that the messages sent back to the clients would be sufficiently informative to track down the error cause.

Testing of the path planner component begins by ensuring that the path planning plugin is located and loaded correctly when the component starts. The necessary condition for correct planner operation is the presence of a valid scene file. A valid scene file contains a syntactically correct description of the geometries and positions of the robot as well as the obstacles present in its workspace. Once the planner is informed of the scene, the description of the planning problem is completed by providing valid start and end positions for the robot's motion. A valid position is any position that can be physically reached by the robot in its environment. An example of an *invalid* position is where the robot is colliding with an obstacle, or a physically unrealizable configuration of the robot joints. The description of the scene and positions involves a lot of numbers and it is not feasible to enter them manually [1]. This is were the testing tool, Parth, comes in. Parth can be used for parsing and visually representing the scene file. Random collision-free configurations of the robot can be generated which can then be used as intermediate points for describing the desired path. These are sent to the framework together with the name of the scene file. In this way, it becomes easy to generate a lot of inputs for testing the path planner. After the path is planned, Parth can get it from the framework and display it visually. The correct working of the planner can thus be verified.

What kind of obstacles should be used during testing? Some scenes for testing of PRM planners are described in [51, 52]. The cage is a complex environment which puts heavy load

[1] In real use, this data will come from the client.

on the collision checker. An example of the Powercube robot arm within a cage obstacle is shown in figure 6.3. Another approach is simply to look at the scene and manually model the environment in which the robot will operate. Modeling can be done with a tool like Blender, after which the scene can be converted into a format suitable for the planner. Figure 6.4 shows the KUKA robot arm within a simple scene where obstacles have been represented with a cylindrical and a cubical bounding boxes.

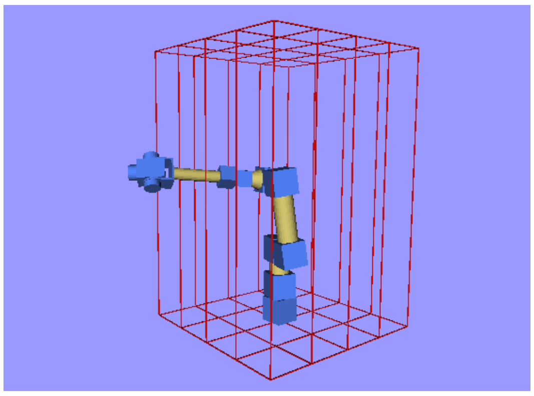


Figure 6.3. A cage obstacle

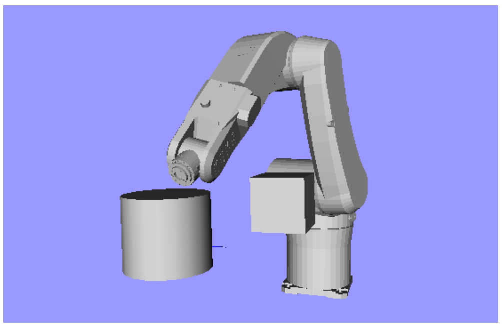


Figure 6.4. Obstacle representation with geometric bounding bo

Using a pair of cameras, it is also possible to generate an automatic representation of the scene. Another option is to use a laser scanner instead of a camera for generating scene data.

The motion controller component heavily depends on the proper function of the sensor and robot components. The sensor component is responsible for providing current values of the robot's position, velocity and acceleration. It does this by querying the robot component, which provides an abstraction to the robot being used.

The functions provided by the robot component are tested using APIease. The process involves setting the target values of joint parameters like positions, velocities and accelerations, then reading them back while the robot is moving. Another important action is to check and set the limits of the joint parameters. In the framework, the joint limits are used to scale the joint parameters within $[0, 1]$ with 0 representing the lower limit and 1 representing the upper limit. This ensures that values never exceed the limits. The correctness of the inverse and forward kinematics solvers is then tested since these functions are invoked whenever the representation of a parameter inside the framework changes from configuration to cartesian space or vice-versa.

Position data from the sensor component was tested by using Parth to query the current position and update the on-screen model. A simple visual check is often enough to determine if the values are incorrect. For finer verification, the joint positions are incremented/decremented by known, small values and sensor values are checked to see if they reflect the changes.

The first test of the motion controller was to see if the robot moved. After the robot seemed to be following the generated path, two motion parameters were tuned. The first parameter affects how accurately the robot tracks the path, which the second parameter affects the smoothness of the motion. Smoothness in this context refers to the decrease in velocity as the robot approaches the current position setpoint. There is a velocity decrease as the setpoint is approached because the position controller typically uses trapezoidal velocity profiling with a velocities set to zero at the motion endpoints. Therefore, the time invariant motion controller feeds the next position setpoint before the downward ramp in the trapezoidal velocity profile begins. If the deceleration is low, the ramp down begins earlier and vice versa. A steeper value of the ramp gives better path tracking accuracy, since the robot can get quite close to the position setpoint before it starts moving to the next setpoint. Finally, communication with the robot was disrupted while the robot is moving. This is intended to verify that the motion controller can respond to abnormal conditions and bring itself into a well-defined error state. The presence of an error state brings the robot to a stop.

One more test at the framework level involves generating various errors as the framework is in operation and ensuring that none of the components block. Components should be responsive to new commands under all circumstances.

6.3 An example use case

This section describes how the Sarathi framework is used at the Computer Vision and Active Perception (CVAP) laboratory at KTH.

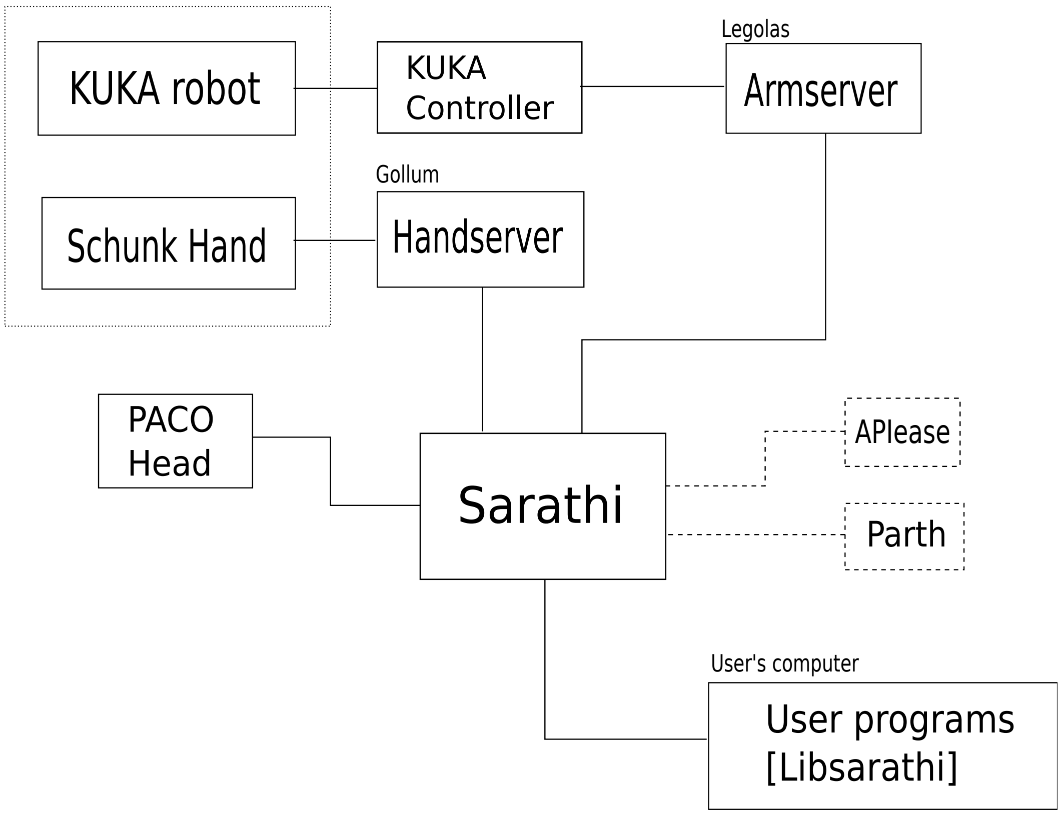


Figure 6.5. Use case layout

KUKA robot is a KUKA KR5 Sixx R850 6dof robot arm

Schunk hand is the Schunk SDH three fingered dexterous hand attached to the robot end effector

KUKA Controller is the proprietary robot controller from KUKA Gmbh (the robot manufacturer)

Hand server is an interface to the Schunk hand, which is developed in-house

Armserver is an interface to the KUKA arm, which is developed in-house

PACO Head is a robotic head containing 4 cameras which is used to observe the robot's workspace

The KUKA controller, armserver, handserver, PACO head, Sarathi and user programs execute on different computers connected to the lab network. Sarathi runs on a different computer not because of computational load considerations but to avoid disrupting the software setup of the existing lab computers.

The robot+hand is used to grasp various objects in the robot workspace. Sarathi is used for moving the robot towards the desired objects without knocking over other objects, hitting the table on which the objects are kept or moving into walls. Therefore, the objects with which collisions are to be avoided are considered to be obstacles. The PACO head is used to observe the scene and software is available to generate three dimensional models of the objects in the workspace. A script has been created to use the PACO head for generating a scene file appropriate for Sarathi to use. A plugin has been created which represents the robot arm and hand as a 13 DOF kinematic chain, and interacts with the hand and arm servers. Thus, although two different interfaces running on two different computers are used for actual communication, a uniform interface is presented by the robot component to the rest of the Sarathi framework.

During operation, the KUKA controller, handserver, armserver and PACO head are initialized. Then, Sarathi is started with the MPK path planning plugin, the time invariant motion controller plugin and the correct robot plugin. Parth and APIease may be running for monitoring purposes, but their execution is optional.

The user programs use libsarathi to interact with the Sarathi server. Typically, the user program uses the PACO head to generate a scene file, then sends a robot destination position to Sarathi. The SOAP server component receives the request and passes it on the Command Processor. The command processor programs the path planner with the current and destination positions, as well as the scene file. The path generated by the planner is then sent to the motion controller, which moves the robot along the path. At all times, the user program can query the status of the framework components. If any errors occur, an error report, including error data, is send back to the client.

Thus, once Sarathi is up and running, the user programs simply need to issue a motion destination and every other aspect of motion planning and control is handled by Sarathi. The same user programs can be run on other lab robots simply by replacing the current robot plugin with the desired robot plugin. Sarathi doesn't even have to be restarted for this to happen.

At the time of this writing, the system is under evaluation.

Chapter 7

Conclusion and future Work

7.1 Conclusion

This thesis commenced by proposing the hypothesis that it is possible to have a generic software framework for robot motion planning and control, which can be configured to solve specific robot motion tasks. In order to test the veracity of the hypothesis, the requirements for the generic framework were identified. Based on these requirements, a component based framework was designed and implemented. The framework was then tested by using it to control three different robotic arms. It is possible to use different algorithms in each framework component through the use of plugins. Therefore, it is possible to adapt the framework to the desired task by writing plugins suitable for the task. Considering the successful tests conducted on the framework, it is established that the proposed hypothesis is valid.

7.2 Future work

Although the framework works satisfactorily and is a sufficient proof of concept, it has not yet been fully developed. This chapter identifies areas where improvements can be made, in order to further enhance the effectiveness and scope of application of the framework.

Of the components described in section 5.1, only the SOAP server, Command Processor, Path Planner, Robot, Sensor and Motion Controller have been implemented. The others need to be written and the inter-component communication needs to be 're-wired' to accommodate the new components.

A study needs to be made for applying the framework structure as it exists to the case of moving obstacles.

The plugins available for the framework are limited. More plugins need to be written in order to use the framework for scenarios beyond those used to prove the concept in this thesis. Availability of a non-holonomic path planning plugin can immediately make the framework applicable to mobile robots.

A testing tool called Parth, provided with the framework, can be used for limited visualization. More generic visualization methods should be adopted. Advantage can be taken of existing tools like Peekabot[16]. Blender for robotics [1, 2] is a promising direction for visualization and robot interaction.

OROCOS, which forms the backbone of the framework has several capabilities which can be exploited to enhance the features of the framework. Scripting support and the ability to distribute the components over different computers would be particularly attractive.

The interfaces of the components as well as the component-plugin interfaces can be made more exhaustive. Currently, only the functions useful for the immediate use case are present.

More functions can be added to the SOAP server-client interface, so that client programs have a greater runtime control over various framework features.

Developer documentation should be created to make it easier for new developers to contribute to the project.

Wider deployment of the framework will undoubtedly reveal more needs. The free and open source nature of the framework makes it easy for anyone to modify the code, fix bugs and contribute their improvements back to the main source tree.

Appendix A

Creating robot models for the MPK path planner

This appendix summarizes the steps taken to create a robot model for the 6 DOF KUKA KR5 Sixx R850 robotic arm.

A.1 Introduction

Robot models for use with the MPK library need to have a "collision model" in Open Inventor's .iv file format. These collision models are basically .iv models of individual robot links. The robot definition (.rob) file contains parameters needed to "assemble" the link models into the entire robot arm. The process of generating a suitable link model in .iv format and finding the "assembly" parameters is not immediately obvious. This document describes the steps taken to build a .rob file for the KUKA KR5 Sixx R850 robot arm.

A.2 Quickstart

1. Generate CAD model of individual links
2. Open each model in blender 3d[1][1]. Save it in blender format
3. Shift origin of global coordinate system of each link to the point about which the link rotates
4. Export model obtained in step 3 to .iv format
5. Use blender to calculate offsets of coordinate system origin of each link from coordinate system origin of previous link.

A.3 Detailed steps

1. Download a CAD model of the robot from KUKA website in .STL format
2. Import the model into blender. I used version 2.48a

[1]Blender is just an example. It is possible to use any software that can load the CAD model and save it in Open Inventor format.

3. Go to edit mode, select all vertices ('A' key), set the limit to 0 and remove doubles. This step makes it easier to select parts of the model which need to be deleted later. Also, it could lead to a smaller .iv format representation (less vertices)

4. Now, we need models for the individual links. Decide which link is needed and the delete all other parts of the model. Save the file as a .blend file. Next, export it to the .iv format. View the resulting .iv file with the ivview program

5. Repeat step 4 until you have .blend and .iv files for all links

6. If you followed exactly the same steps while generating the link models for all links, the resulting .iv file models will be to scale and the individual link models will be positioned at the right places. This means that if you collect together all the .iv link models in a single .iv file (see box to see how), you'll see the entire robot arm with all the links in their right places

```
    #Inventor V2.1 ascii
    #robot model
    Separator
    {

        File {name "kuka_0.iv"}
        File {name "kuka_1.iv"}
        File {name "kuka_2.iv"}
        File {name "kuka_3.iv"}
        File {name "kuka_4.iv"}
        File {name "kuka_5.iv"}
        File {name "kuka_6.iv"}

    }
```

7. Next, we need to position the local coordinate system of the link models (in the exported .iv files) at the right point and orientation. This is important because in the robot definition files, the joint rotations are specified around the axes of the local coordinate system. Unless the local coordinate system is in the right place in each link model, your robot links will go all over the place when you use the .rob file in MPK

8. The local coordinate system in the exported .iv file matches the global coordinate system in blender. Thus, you need to move the global coordinate system to that point on a link, about which the link rotates, when connected to the previous link. See figure A.1 to see where I had to position the global coordinate system for each link of my robot. Once you have shifted the global coordinate system to the right place, save the .blend model and export it to .iv format again

9. Now, if you use the code given in the box above to collect the links together into a robot model, its not going to work. This is because inventor will place all the links such that the origins of their local coordinate system coincide. You need to specify the offset of the origin of the coordinate system of each link, from the origin of the coordinate system of the previous link. For example, lets say we need to find the offset of link 3 w.r.t link 2. To do this, open the .blend model of link 2. Zoom in to the point on the link, where link 3's coordinate system origin will touch, when it is

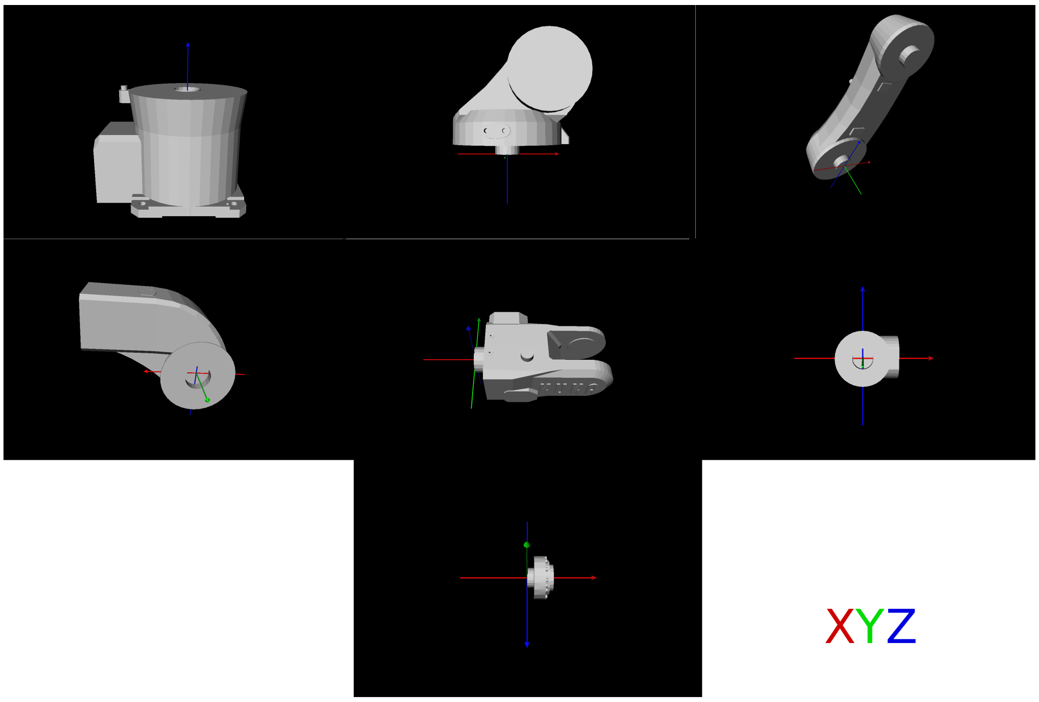


Figure A.1. Link models

properly assembled. Get the coordinates of this point. These will be the offset of link 3 w.r.t link 2. Note this data carefully. You'll need it to build the .rob file

10. Repeat the steps of point 9 till you know the offset of each link w.r.t its previous link

11. With the data from point 10 and the .iv models from point 8 you can now assemble the coordinate system shifted .iv models into a complete robot model. The same data is used to create a .rob file. Example robot files are distributed with Sarathi

A.4 Coordinate system visualization

It is often useful to insert 3 arrows (the X-, Y- and Z- axes) to depict the coordinate system in your .iv model. Use the code below in your .iv file to get an idea of where the origin lies and how the coordinate system is oriented.

```
#Coordinate system visualization
Separator
{
#Y-axis
Separator
{
```

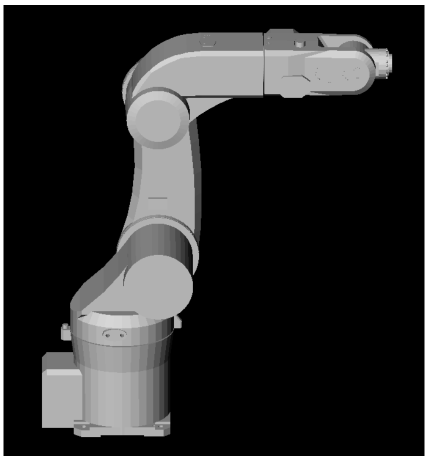


Figure A.2. The assembled KUKA robot

```
            BaseColor {rgb 0 1 0}
            Rotation {rotation 0 0 0 -1.5707}
            Cylinder {
              parts    ALL   # SFBitMask
              radius 1       # SFFloat
              height 200       # SFFloat
         }
         Translation {translation 0 100 0}
         Cone {
              bottomRadius 4
              height 8
         }
    }
    #X-axis
    Separator
    {
            BaseColor {rgb 1 0 0}
            Rotation {rotation 0 0 -1 1.5707}
            Cylinder {
              parts    ALL   # SFBitMask
              radius 1       # SFFloat
              height 200       # SFFloat
         }
         Translation {translation 0 100 0}
         Cone {
              bottomRadius 4
              height 8
         }
    }
    #Z-axis
```

```
Separator
{
        BaseColor {rgb 0 0 1}
        Rotation {rotation 1 0 0 1.5707}
        Cylinder {
          parts    ALL   # SFBitMask
          radius 1       # SFFloat
          height 200       # SFFloat
     }
     Translation {translation 0 100 0}
     Cone {
          bottomRadius 4
          height 8
     }
}
}
```

Appendix B

Describing scenes for the MPK path planner

The MPK path planner module in the Sarathi framework needs a 'scene file' as input. A 'scene file' is a text file describing the environment in which the path planner operates. The environment consists of a robot and the obstacles in its workspace. The environment description thus consists of a description (including location) of the robot and the obstacles. This document explains how to create a custom scene file describing the environment of your application.

B.1 Introduction

The MPK scene file format is an extension of the Open Inventor scene file format. MPK extends the Inventor file format by three node types: mpkObstacle, mpkRobot and mpkIncludeFile. These are derived from the Inventor node type SoSeparator and can thus have the same attributes as a SoSeparator, in addition to their own specific attributes.

In this appendix, we will create a custom scene file to which we will add the KUKA KR5 Sixx R850 robot by using the mpkRobot node. Next, we will add obstacles, using the mpkObstacle and mpkIncludeFile nodes.

B.2 Quick summary

1. Open empty text file
2. Add the line: "#Inventor V2.0 ascii" (without the "") as the first line
3. Add a robot using mpkRobot node
4. Add obstacles using either mpkObstacle or read them in from other files using mpkIncludeFile
5. Save file

B.3 Detailed steps

B.3.1 Understanding the coordinate system of your robot model

Before you create the scene file, it is important to know the orientation and location of the origin of the coordinate system of your robot model. For the KUKA KR5 Sixx R850 robot model referred to in this document, the origin is located as shown in figure B.1. (This figure shows a part of the robot base only, to make the illustration more clear)

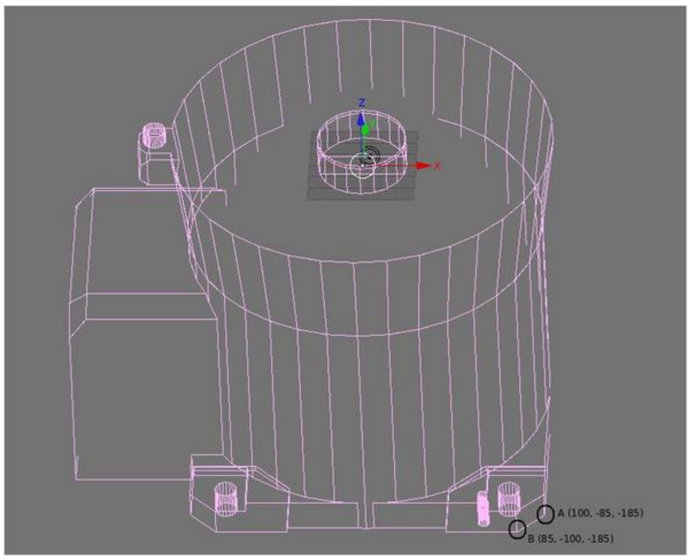


Figure B.1. KUKA KR5 Sixx R850 coordinate system location

Within this coordinate system, the points A and B at the edge of the robot base have coordinates as shown in figure B.1. When the robot model is inserted into the scene description, this coordinate system will be the global coordinate system for the scene description. However, it is not very convenient to have the origin of the global coordinate system at a physically inaccessible point inside the robot's base. Hence, when adding the robot to the scene description, we will translate it such that the global coordinate system is located at the (more physically accessible) point A (it could just as well be at point B, or at any other point whose coordinates are known).

B.3.2 Creating the scene file

1. Open an empty text document using your favourite text editor. Save it with a .iv extension. For example, mykukascene.iv

2. Since the scene description is an Open Inventor file, the first line must be
```
#Inventor V2.0 ascii
```

3. Comments can be added using a #at the start of a file
```
#Inventor V2.0 ascii
# This is a comment
# Scene file with kuka robot and 3 obstacles
```

4. Add lines describing the kuka robot
```
DEF robot mpkRobot {
fileName "kuka.rob"
}
```
The file kuka.rob contains a model of the kuka robot. This file is provided for you in the Sarathi distribution.

5. If we add the robot as described above, the global coordinate system is located at a physically inaccessible point within the base of the robot, as shown in figure1 . To shift the location of the global coordinate system to point A, we add a line specifying a translation of the robot. Our revised lines describing the kuka robot are

```
DEF robot mpkRobot {
fileName "kuka.rob"
translation -100 85 185
}
```

   We are moving the robot with respect to the global coordinate system, which is now effectively at the point A. See figure B.2

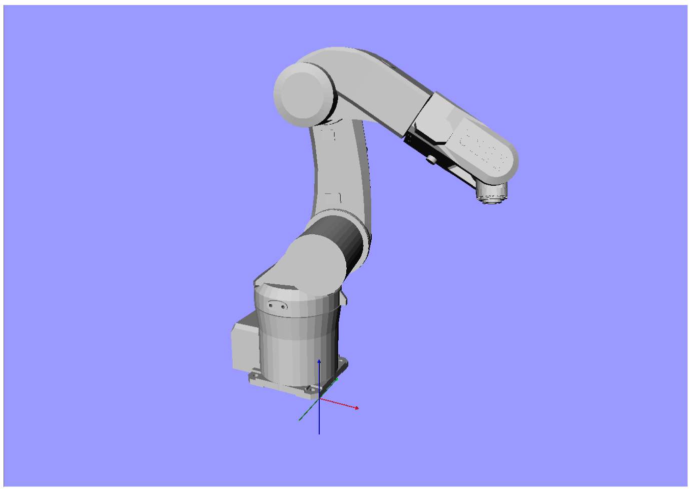


Figure B.2. Robot translated to reposition the global coordinate system

6. We now add obstacles using mpkObstacle. An obstacle can be any Open Inventor object node type. We will add a cube with width=height=depth=120mm. (Note that the openinventor Cube type doesn't have to be a geometric cube. This means that the width, height and depth parameters need not be equal to each other.) By default, the cube is created such that it is centered at the origin of the global coordinate system. Let us position our cube such that it is 150mm away from the origin of the global coordinate system along the X- and Y- directions

```
DEF box mpkObstacle {
    Separator {
        Translation {translation 150 150 0}
        DEF __triangulate__ Cube {
            width 120
            height 120
            depth 120
            }
        }
}
```

   The 'DEF __triangulate__' tag is required to tell MPK that the triangles of the Cube node are part of the collision model. Similarly, one could add other Inventor models inside the same mpkObstacle node. Without the 'DEF __triangulate__' tag, the models would be displayed but will not be checked for collisions. The syntax for translation is: Translation {translate x y z} where x,y and z refer to translation distances along those axes. Consecuting translation transforms are concatenated. The result of the above obstacle addition is shown in figure B.3

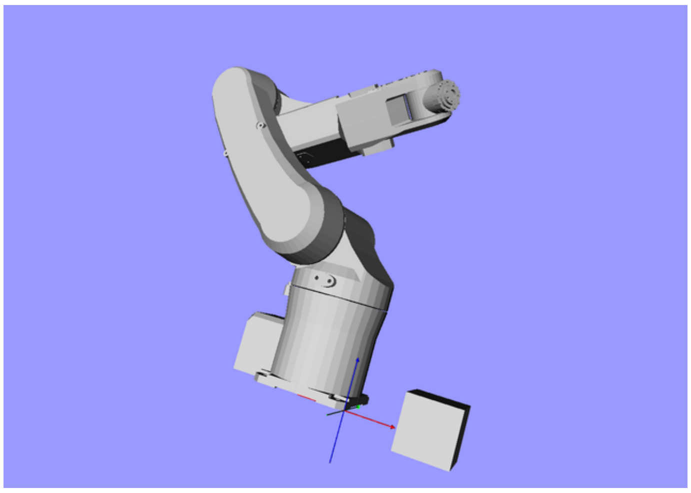


Figure B.3. Cube added as obstacle

7. Notice how the cube appears to go below the plane of the robot base in figure B.3. This is because, the cube is, by default, centered at the origin of the global coordinate system. We merely translated it along the X- and Y- axes in the previous step. To make the cube base coplanar with the robot base, we need to translate it upwards (along the positive Z- axis in this case) by half its height. Thus, we change the translation parameters line to: Translation {translation 150 150 60}. The result is shown in figure B.4.

8. We will now add a cylindrical obstacle. By default, Open Inventor adds a cylindrical obstacle such that it is centered at the origin of the global coordinate system and its axis is aligned with the Y axis. (Ignore the commented line. It is explained in the next step)

```
    DEF cyl mpkObstacle {
        Separator {
            Translation {translation 150 150 500}
    #       Rotation {rotation 1 0 0 -1.57}
            DEF __triangulate__ Cylinder {
                parts   ALL
                radius  120
                height  200
                }
            }
    }
```

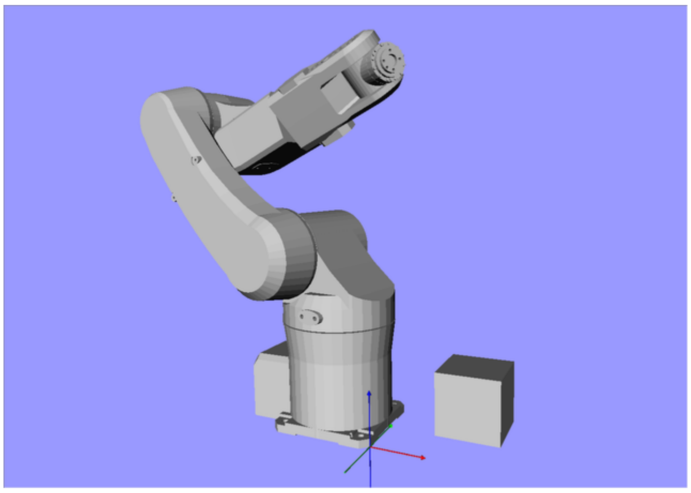


Figure B.4. Cube coplanar with robot base

The result is shown in figure B.5.

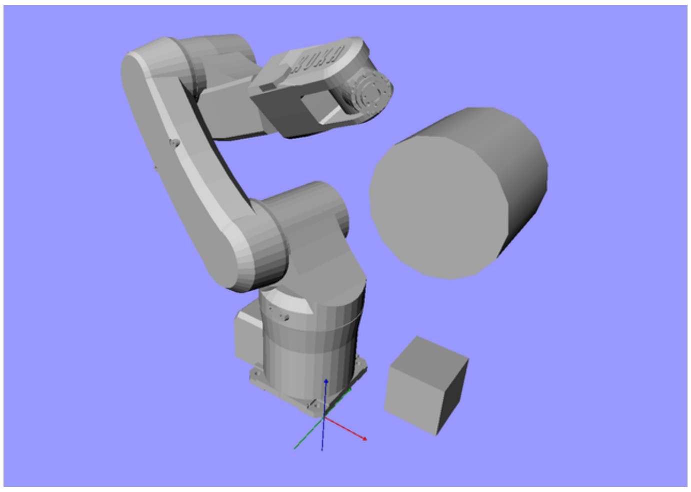


Figure B.5. Cylindrical obstacle added to the scene

9. What if we want to rotate the cylinder by 90 degrees? Rotations are specified by a line like: Rotation rotation {x y z angle}, where one of x,y or z must be 1, indicating the axis around which rotation should take place. angle is specified in radian. Consecutive rotation transforms are concatenated. So to rotate the cylinder about the X- axis by -90 degrees, just uncomment the Rotation... line in the code above. The result is shown in figure B.6.

10. Now let us put the robot on a table. We will consider a table to be an open inventor "Cube" with relatively large dimensions along two axes (length and width) and a very

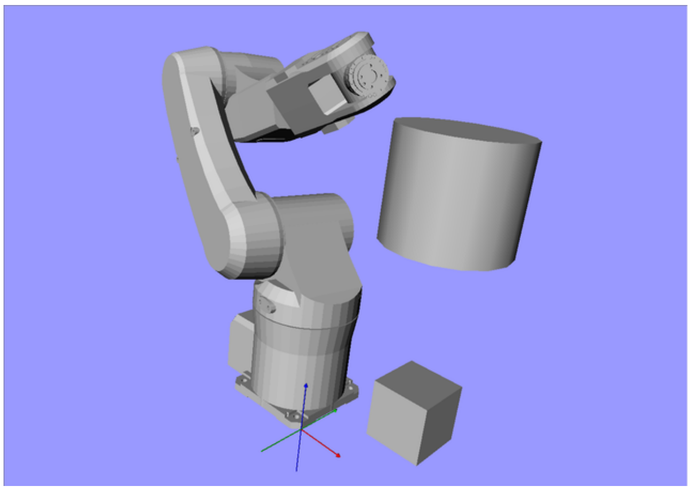


Figure B.6. Rotated cylindrical obstacle

small dimension along the third axis (the thickness). We can add it similar to the way we added the Cube previously. However, this time we do things a bit differently. Assume that the table is described in a file called “table.iv”

```
     DEF table mpkObstacle {
         Separator {
             Scale {scaleFactor 1 1 1}
             DEF __triangulate__ File {name "table.iv"}
         }
     }
```

Note that the Scale {scaleFactor 1 1 1} line has no effect here, since it just scales the table by a factor of 1 along all axes. It is included here to show the possibility of scaling the object being included. The table.iv file is simply contains

```
     #Inventor V2.0 ascii
     Separator {
             Translation {translation 0 0 -5}
             Cube {
                 width 1200
                 height 1200
                 depth 10
             }
     }
```

The result is shown in figure B.7

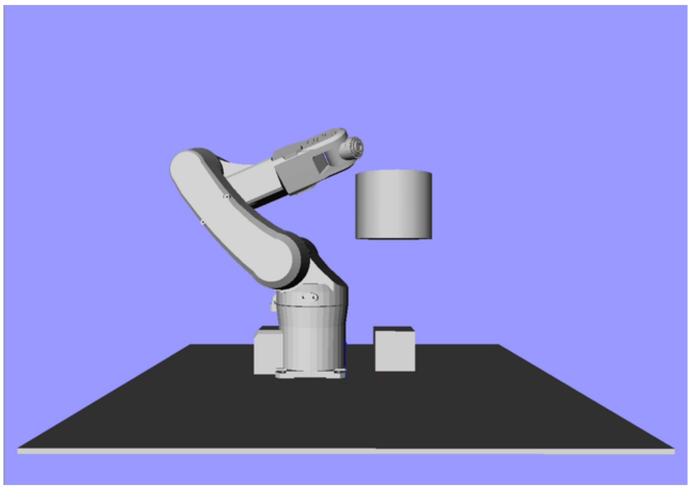


Figure B.7. Robot with table

B.3.3 The complete scene description file

```
#Inventor V2.0 ascii
#Scene file with KUKA robot and 3 obstacles
DEF robot mpkRobot {
    fileName "kuka.rob"
    translation -100 85 185
#    rotation 1 0 0 -1.57
}
DEF box mpkObstacle {
    Separator {
        Translation {translation 150 150 60}
        DEF __triangulate__ Cube {
            width 120
            height 120
            depth 120
            }
        }
}
DEF cyl mpkObstacle {
    Separator {
        Translation {translation 150 150 500}
        Rotation {rotation 1 0 0 -1.57}
        DEF __triangulate__ Cylinder {
            parts   ALL
            radius  120
            height  200
            }
        }
}
DEF table mpkObstacle {
    Separator {
        Scale {scaleFactor 1 1 1}
        DEF __triangulate__ File {name "table.iv"}
    }
}
```

Bibliography

[1] Blender. URL http://www.blender.org/. Cited on pages 42 and 44.

[2] Blender for robotics. URL http://wiki.blender.org/index.php/Robotics:Index. Cited on page 42.

[3] Common object request broker architecture. URL http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture. Cited on page 28.

[4] Gnu plotutils. URL http://www.gnu.org/software/plotutils/. Cited on page 29.

[5] The gsoap toolkit for soap web services and xml-based applications. URL http://www.cs.fsu.edu/~engelen/soap.html. Cited on page 28.

[6] How to create qt plugins. http://doc.trolltech.com/4.5/plugins-howto.html. URL http://doc.trolltech.com/4.5/plugins-howto.html. Cited on page 27.

[7] The internet communications engine. URL http://www.zeroc.com/. Cited on page 28.

[8] Kuka kr 5 sixx r850 robotic arm. URL http://www.kuka-robotics.com/en/products/industrial_robots/small_robots/kr5_sixx_r850/. Cited on pages 29 and 33.

[9] The libhyperpoint api documentation. URL http://cogvis.nada.kth.se/~behere/sarathi/docs/libhyperpoint/html/annotated.html. Cited on page 34.

[10] Motion planning kernel (mpk). URL http://ramp.ensc.sfu.ca/mpk/index.html. Cited on page 32.

[11] Motion planning kit. URL http://robotics.stanford.edu/~mitul/mpk/index.html. Cited on page 32.

[12] The motion strategy library. URL http://msl.cs.uiuc.edu/msl/index.html. Cited on page 31.

[13] Oopsmp. URL http://www.kavrakilab.org/OOPSMP/index.html. Cited on page 32.

[14] The open inventor toolkit. URL http://oss.sgi.com/projects/inventor/. Cited on page 32.

[15] Openrave. URL http://openrave.programmingvision.com/. Cited on pages 5 and 32.

[16] Peekabot. URL http://www.peekabot.org/. Cited on page 42.

[17] The player project. URL http://playerstage.sourceforge.net/. Cited on page 5.

[18] The qt network module. URL http://doc.trolltech.com/4.5/qtnetwork.html. Cited on page 28.

[19] Robwork. URL http://www.robwork.org. Cited on page 5.

[20] Ros. URL http://www.ros.org. Cited on page 5.

[21] The schunk dextrous hand. URL http://www.schunk-modular-robotics.com/left-navigation/service-robotics/components/actuators/robotic-hands/sdh.html. Cited on page 33.

[22] Software framework. URL http://en.wikipedia.org/wiki/Software_framework. Cited on page 2.

[23] Xml rpc. URL http://www.xmlrpc.com/. Cited on page 28.

[24] *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks*, 2002. URL http://www.cs.fsu.edu/~engelen/ccgrid.pdf. Cited on page 27.

[25] René Zapata Abraham Sánchez López and Maria A. Osorio Lama. Sampling-based motion planning: A survey. 2008. URL http://www.cic.ipn.mx/portalCIC/s11/vol12-01/v12no1_Art01.pdf. Cited on page 10.

[26] Nancy M. Amato and Yan Wu. A randomized roadmap method for path and manipulation planning. In *In IEEE Int. Conf. Robot. & Autom*, pages 113–120, 1996. Cited on page 10.

[27] Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Rajeev Motwani, Tsai-Yen Li, and Prabhakar Raghavan. A random sampling scheme for path planning. *Int. J. Rob. Res.*, 16(6):759–774, 1997. ISSN 0278-3649. Cited on page 10.

[28] Sagar Behere. Creating robot models for the mpk path planner. URL http://cogvis.nada.kth.se/~behere/sarathi/dokuwiki/doku.php?id=creating_robot_models_for_the_mpk_path_planner. Cited on page 32.

[29] Sagar Behere. Describing scenes for the mpk path planner. URL http://cogvis.nada.kth.se/~behere/sarathi/dokuwiki/doku.php?id=describing_scenes_for_the_mpk_path_planner. Cited on page 32.

[30] Sagar Behere. Libsarathi api documentation. URL http://cogvis.nada.kth.se/~behere/sarathi/docs/libsarathi/html/annotated.html. Cited on page 34.

[31] Sagar Behere. The libsarathi usage guide. URL http://cogvis.nada.kth.se/~behere/sarathi/dokuwiki/doku.php?id=libsarathi_usage_guide. Cited on page 34.

[32] Luigi Biagiotti and Claudio Melchiorri. *Trajectory Planning for Automatic Machines and Robots*. Springer Publishing Company, Incorporated, 2008. ISBN 3540856285, 9783540856283. Cited on pages 8, 12, and 13.

[33] James E. Bobrow, S. Dubowsky, and J.S. Gibson. Time optimal control of robotic manipulators along specified paths. *The International Journal of Robotics Research*, 1985. Cited on page 16.

[34] R. Bohlin and Lydia E. Kavraki. Path planning using lazy prm. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 521–528, San Fransisco, CA, April 2000. IEEE Press, IEEE Press. Cited on page 11.

[35] D. Box. Simple object access protocol 1.1. `http://www.w3.org/TR/SOAP`, 2000. Cited on page 28.

[36] Michael S. Branicky, Ross Alan Knepper, and James Kuffner. Path and trajectory diversity: Theory and algorithms. In *International Conference on Robotics and Automation*. IEEE RAS, May 2008. Cited on page 11.

[37] Herman Bruyninckx. Open robot control software: the OROCOS project. In *IEEE Int. Conf. Robotics and Automation*, pages 2523–2528, 2001. Cited on page 22.

[38] Herman Bruyninckx. Bayesian probability, 2002. URL `http://www.mech.kuleuven.ac.be/~bruyninc/pubs/urks.pdf`. Cited on page 17.

[39] Herman Bruyninckx. *OROCOS: design and implementation of a robot control software framework*, 2002. URL `http://www.cs.jhu.edu/~hager/Public/ICRAtutorial/Bruyninckx-OROCOS/icra2002-tut.pdf`. Cited on pages 18 and 20.

[40] Herman Bruyninckx. Bayesian probability theory, September 2005. URL `http://www.roble.info/basicST/stat/pdf/Bayes-1.pdf`. Cited on page 17.

[41] Herman Bruyninckx. Open RObot COntrol Software. `http://www.orocos.org/`, 2008. URL `http://www.orocos.org`. Cited on page 22.

[42] Herman Bruyninckx and Peter Soetens. Generic real-time infrastructure for signal acquisition, generation and processing. URL `http://www.mech.kuleuven.ac.be/~bruyninc/pubs/rtlab-design.pdf`. Cited on page 17.

[43] Herman Bruyninckx, Peter Soetens, and Bob Koninckx. The real-time motion control core of the Orocos project. In *IEEE Int. Conf. Robotics and Automation*, pages 2766–2771, 2003. Cited on page 22.

[44] B. Cao, G.I. Dodds, and G.W. Irwin. Constrained time-efficient and smooth cubic spline trajectory generation for industrial robots. *IEE Proceedings -Control Theory and Applications*, 144(5):467–475, 1997. ISSN 1350-2379. Cited on page 13.

[45] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005. Cited on pages 7, 9, and 10.

[46] D. Constantinescu and E.A. Croft. Smooth and time-optimal trajectory planning for industrial manipulators along specified paths. In *AMSE Dynamic Systems and Control Division*, 1999. URL `http://www.me.uvic.ca/~danielac/constantinescu_imece99.pdf`. Cited on page 13.

[47] Daniela Constantinescu. Smooth time optimal trajectory planning for industrial manipulators. Master's thesis, Transilvania University, 1995. Cited on page 13.

[48] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201095289. Cited on pages 8, 12, and 16.

[49] O. Dahl. Path-constrained robot control with limited torques-experimental evaluation. 10(5):658–669, 1994. ISSN 1042-296X. Cited on page 16.

[50] P.J. Davis. *Interpolation and Approximation*. Dover, 1976. Cited on page 12.

[51] Roland Geraerts and Mark H. Overmars. A comparative study of probabilistic roadmap planners. Technical Report UU-CS-2002-041, Department of Information and Computing Sciences, Utrecht University, 2002. Cited on pages 10 and 38.

[52] Roland Geraerts and Mark H. Overmars. Sampling techniques for probabilistic roadmap planners. Technical Report UU-CS-2003-041, Department of Information and Computing Sciences, Utrecht University, 2003. Cited on pages 10 and 38.

[53] Roland Jan Geraerts. *Sampling-based Motion Planning: Analysis and Path Quality*. PhD thesis, Utrecht University, 2006. URL `http://people.cs.uu.nl/roland/pdf/thesis_lowres.pdf`. Cited on page 10.

[54] Donald H. House. Splines. URL `http://www.cs.clemson.edu/~dhouse/courses/405/notes/splines.pdf`. Cited on page 29.

[55] David Hsu, Jean-Claude Latombe, and Hanna Kurniawati. On the probabilistic foundations of probabilistic roadmap planning. *Int. J. Rob. Res.*, 25(7):627–643, 2006. ISSN 0278-3649. Cited on page 10.

[56] James J. Kuffner Jr. and Steven M. Lavalle. Rrt-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA*, pages 995–1001, 2000. Cited on page 11.

[57] 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. Cited on page 17.

[58] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. 12(4):566–580, 1996. ISSN 1042-296X. Cited on page 10.

[59] Lydia Kavraki, Mihail N. Kolountzakis, and Jean-Claude Latombe. Analysis of probabilistic roadmaps for path planning, 1998. Cited on page 10.

[60] Lydia Kavraki and Jean-Claude Latombe. Randomized preprocessing of configuration space for fast path planning. In *IN PROC. IEEE INTERNAT. CONF. ROBOT. AUTOM. (ICRA*, pages 2138–2145, 1994. Cited on page 10.

[61] Lydia E. Kavraki. *Random networks in configuration space for fast path planning*. PhD thesis, Stanford, CA, USA, 1995. Cited on page 10.

[62] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998. Cited on page 11.

[63] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521862051. Cited on page 7.

[64] Steven M. Lavalle, James J. Kuffner, and Jr. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2000. Cited on page 11.

[65] T. Lozano-Perez. Spatial planning: A configuration space approach. C-32(2):108–120, 1983. ISSN 0018-9340. Cited on page 9.

[66] Peter S. Maybeck. *Stochastic models, estimation, and control*, volume 141 of *Mathematics in Science and Engineering*. Academic Press, Inc., 1979. URL http://www.cs.unc.edu/~welch/media/pdf/maybeck_ch1.pdf. Cited on page 17.

[67] S. Munir and W.J. Book. Internet based teleoperation using wave variables with prediction. In *Proc. IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, volume 1, pages 43–50 vol.1, 2001. Cited on page 34.

[68] Mark H. Overmars and Mark H. Overmars T. A random approach to motion planning. Technical report, 1992. Cited on page 10.

[69] F. Pfeiffer and R. Johanni. A concept for manipulator trajectory planning. 3(2): 115–123, 1987. ISSN 0882-4967. Cited on page 16.

[70] G.M. Phillips. *Interpolation and Approximation by Polynomials*. Springer, 2003. Cited on page 12.

[71] Eric S Raymond. *The Art of Unix Programming*. Addison Wesley, 2003. URL http://www.catb.org/~esr/writings/taoup/html/index.html. Cited on page 21.

[72] Gildardo Sánchez and Jean claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *In Int. Symp. Robotics Research*, pages 403–417, 2001. URL http://robotics.stanford.edu/~latombe/papers/isrr01/spinger/latombe.pdf. Cited on pages 11 and 32.

[73] M. Latombe J.-C. Schwarzer, F. Saha. Adaptive dynamic collision checking for single and multiple articulated robots in complex environments. In *Robotics, IEEE Transactions on*. IEEE, 2005. URL http://robotics.stanford.edu/~latombe/papers/adaptive-bisection/paper.ps. Cited on pages 10 and 32.

[74] Kang Shin and N. McKay. Minimum-time control of robotic manipulators with geometric path constraints. 30(6):531–541, 1985. ISSN 0018-9286. Cited on page 16.

[75] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008. ISBN 1846286417, 9781846286414. Cited on pages 8, 10, 11, 12, 13, and 16.

[76] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley-Interscience, 2006. ISBN 0471708585. Cited on page 17.

[77] J.-J.E. Slotine and H.S. Yang. Improving the efficiency of time-optimal path-following algorithms. 5(1):118–124, 1989. ISSN 1042-296X. Cited on page 16.

[78] Christian Smith. *Input Estimation for Teleoperation*. PhD thesis, Royal Institite of Technology, Stockholm, Sweden, Nov 2009. URL http://www.csc.kth.se/~ccs/Publications/phd_thesis_smith.pdf. Cited on page 34.

[79] Peter Soetens. The orocos component builder's manual. http://www.orocos.org/stable/documentation/rtt/v1.10.x/doc-xml/orocos-components-manual.html, 2007. URL http://www.orocos.org/stable/documentation/rtt/v1.10.x/doc-xml/orocos-components-manual.html. Cited on pages 24, 26, and 27.

[80] P. Svestka. *Robot Motion Planning Using Probabilistic Road Maps*. PhD thesis, Utrecht University, 1997. Cited on page 10.

[81] Antonio Visioli. Trajectory planning of robot manipulators by using algebraic and trigonometric splines. *Robotica*, 18(6):611–631, 2000. ISSN 0263-5747. Cited on page 13.

[82] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995. URL `http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf`. Cited on page 17.

[83] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. ISBN 0201624958. Cited on page 32.

[84] ISO/IEC JTC1 SC22 WG21. Iso/iec tr 18015: Technical report on c++ performance. Technical report, ISO, July 2004. URL `http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf`. Cited on page 22.

[85] Zhaoxue Yang and Edward Red. On-line cartesian trajectory control of mechanisms along complex curves. *Robotica*, 15(3):263–274, 1997. ISSN 0263-5747. Cited on page 8.

[86] Milos Zefran. Review of the literature on time-optimal control of robotic manipulators. Cited on page 16.

[87] C.S. Zhao, M. Farooq, and M.M. Bayoumi. Analytical solution for configuration space obstacle computation and representation. In *Proc. IEEE IECON 21st International Conference on Industrial Electronics, Control, and Instrumentation*, volume 2, pages 1278–1283 vol.2, 1995. Cited on page 9.

TRITA-CSC-E 2010:014
ISRN-KTH/CSC/E--10/014--SE
ISSN-1653-5715