# Motion Planning and Control of Robot Manipulators

Sergey Pluzhnikov

Embedded Computing Systems
Submission date: June 2012
Supervisor: Geir Mathisen, ITK
Co-supervisor: Aksel Transeth, SINTEF

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU

Norwegian University of Science and Technology

Master's thesis

# Motion Planning and Control of Robot Manipulators

Student: Pluzhnikov Sergey

Academic supervisor: Geir Mathisen

**Spring Semester**

**2012**

**NTNU**
**Norges teknisk-naturvitenskapelige universitet**

**Fakultet for informasjonsteknologi, matematikk og elektroteknikk**
**Institutt for teknisk kybernetikk**

# MASTER THESIS

| | |
|---|---|
| Kandidatens navn: | **Sergey Pluzhnikov** |
| Fag: | **Engineering cybernetics** |
| Oppgavens tittel (norsk): | **Bevegelsesplanlegging og regulering av robotmanipulatorer** |
| Oppgavens tittel (engelsk): | **Motion planning and control of robot manipulators** |

-

Bakgrunn:

Robot motion planning systems can be used to plan collision-free paths between start and end robot manipulator configurations. Many motion planners assume that the robot manipulator works in a static environment. However, many situations (such as e.g. working close to humans) require that the robot can account for dynamic environments. A motion planner with online-planning capabilities is the topic of this thesis assignment.

Tasks:

1. Perform a literature survey on
    - State-of-the-art sample-based and feedback-based motion planning techniques.
    - Pay particular attention to relevant aspects of the methods proposed by [1], [2] and [3].

2. Develop an online motion planner for robot manipulators as a combined approach based on [1], [2] and [3]. Employ the methods from [1] and [3] for offline processing and online collision checking. Employ the methods from [2] for online motion control with task constraints.
3. Implement the motion planner in simulation for a Schunk LWA robot manipulator arm.
4. Analyse the behaviour and performance of the implemented system.
    - Create a test suite which demonstrates key functionality of the motion planner.

[1] T. Kunz, U. Reiser, M. Stilman, A. Verl, "Real-time path planning for a robot arm in changing environments", Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2010.

[2] Y. Yang and O. Brock, "Elastic roadmaps – motion generation for autonomous mobile manipulation", Autonomous Robots 28, 2010.

[3] P. Leven and S. Hutchinson, "A framework for real-time path planning in changing environments", The international journal of robotics research, 2002.

| | |
|---|---|
| Oppgaven gitt: | 09.01.2012 |
| Besvarelsen leveres: | 04.06.2012 |
| Besvarelsen levert: | Utført ved Institutt for teknisk kybernetikk |
| Veileder: | Forsker Aksel A. Transeth, SINTEF Anvendt Kybernetikk |

Trondheim, den 09.01.2012

Geir Mathisen

Faglærer

# Abstract

When a robot performs a task in an unstructured dynamic environment, it has to account for many factors. It should not only keep the track of where it is and how it should move, but also ensure that the kinematic, dynamic and task specific limitations are observed. It is also important that the robot can effectively avoid collisions with static and moving obstacles. In the current thesis we present design and implementation of an algorithm capable to face all these challenges. The system combines principles of *dynamic roadmaps* and *elastic roadmaps* frameworks, both of which are the state-of-art approaches to motion planning problem.

The suggested solution is presented in the context of a broad overview of the literature in motion planning domain focusing on methodology of sample-based and feedback planning in dynamic environments. The implemented algorithm is applied to a 7-degree-of-freedom manipulator and is demonstrated and analyzed through a variety of simulated test scenarios. The result is an extensible and future-oriented planning system that can plan and execute movement between a starting and target position while preserving task constraints and reacting to environment changes in real time.

# Table of contents

# List of Figures

## List of Tables

# 1. Introduction

## 1.1. Motivation

Nowadays more and more robots face the task of working in unstructured dynamic environments. The robots leave specially designated perfectly known working areas and fulfil their tasks in places that have not been specially prepared. This can be a living room in a flat for an assistant robot or a factory floor where a robot has to manoeuvre among people and machines. Operation in such environments is a challenge for modern systems because of their complexity and inherent uncertainty.

According to [**1**], the main difficulties that a robot faces while working in human environments are the following:

- In unstructured environments, a robot can only possess partial knowledge of its surroundings, objects can change their state unbeknownst to the robot, and manipulation tasks may require the end effector to move on a constrained trajectory rather than simply to reach a specific location;
- Complex task requirements impose strict requirements for high-frequency feedback;
- Perception systems of robots have to address an intractable amount of information acquired by multiple sensor modalities;
- Interpretation of the acquired information is complex task as the sensors provide ambiguous and redundant information and introduce uncertainty due to noise;
- In order to avoid collisions robots must solve the high-dimensional problem of distinguishing between objects and free space, calculating how far away objects are, figuring out how they are positioned etc.

In such situation adequate motion planning becomes a crucial requirement for robot operation in dynamic environments.

## 1.2. Background

This section gives a short introduction to the most important background of the thesis. Much more details will be presented in the further chapters.

The goal of the current project is to design and implement a framework that would allow performing motion planning and control of a manipulator in a dynamic environment while preserving task constraints. An example of manipulation task suitable for the framework is movement of a glass of water from one point to another while keeping it upright. Another situation when such framework is needed is when a camera is installed at manipulator’s end effector and it should aim at the same object during movement. Thus, not only position of the end effector, but also its orientation should be considered throughout planning and path following. A task may also constrain position of the end effector, e.g. it can be required to keep the end effector at a specified height above the ground.

To the author's best knowledge, the only planners that consider task constraints (constrains on position and/or orientation of robot's end effector) are *elastic stripes* and *elastic roadmaps* (refer to the Literature survey, sections 2.4.6 and 2.5 for more details). Both frameworks utilize task-level controller - a convenient and powerful tool for generating multi-objective behaviour for robotic systems designed by Khatib and Sentis ( [**2**], [**3**]). Therefore, it looks logical to use the task-level controller for the current project as well.

Last semester in a specialization project the *elastic roadmaps* framework for a mobile manipulator was implemented ( [**4**]) and several drawbacks inherent to the approach were identified as a result of the project (see chapter 2.5). To avoid these shortcomings it was decided to modify some of the framework's functionality. A possibility studied in this project is the integration of the task-level controller into a conventional motion planner to provide fulfilling of task constraints throughout path execution. It is also important that the resulting framework would preserve the opportunity to specify manipulation tasks directly in workspace coordinates, assuming that the configuration space is usually out of interest of robot end users. As an appropriate candidate for this goal the *dynamic roadmaps* (detailed description is given in section 2.6) were chosen.

## 1.3. Contribution

In the current thesis, a broad overview of motion planning methods focusing on sample-based and feedback planning as well as planning in dynamic environments is presented first. Then we demonstrate how *dynamic roadmaps* and *elastic roadmaps* can be merged together. The resulting motion planning and control framework allows performing of complex manipulation tasks with consideration of positional and orientation task constraints during the path execution. The proposed solution exploits the data computed in the offline stage to perform planning and to recognize quickly configurations of the robot that are in collision with obstacles. The planner identifies a path for the end effector that can be followed in at least approximately task-consistent manner. The computed path then is executed with the help of a task-level controller.

The resulting framework was specially designed to control a 7-degree-of-freedom manipulator (Fig. 1). It was implemented in C++ and its performance was tested in multiple scenarios. The results of the simulations demonstrated that the suggested system was capable to solve certain complex manipulation tasks in dynamic environments. However, the cost for maintaining task constraints is the completeness of the proposed method. This issue was thoroughly analyzed and some measurements addressing this problem were suggested.

Some components of the task-level controller were adopted from the project finished last semester ( [**4**]), but a fair amount of design efforts were still needed to adapt it to the current project. More specific details about this topic will be given in the further chapters of the report.

## 1.4. Outline

The structure of the current thesis is the following: a comprehensive literature survey of motion planning techniques with the focus on sampling-based and feedback planning in static and dynamic environments is given first; then the *elastic roadmaps* and the *dynamic roadmaps* are described in details; chapter 3 presents the overall structure and the design of components of the proposed solution; chapter 4 gives some details about the implementation of the framework; in chapter 5 simulation results are presented; chapter 6 discusses the overall performance of the implemented system; topics for future work are suggested in chapter 7.

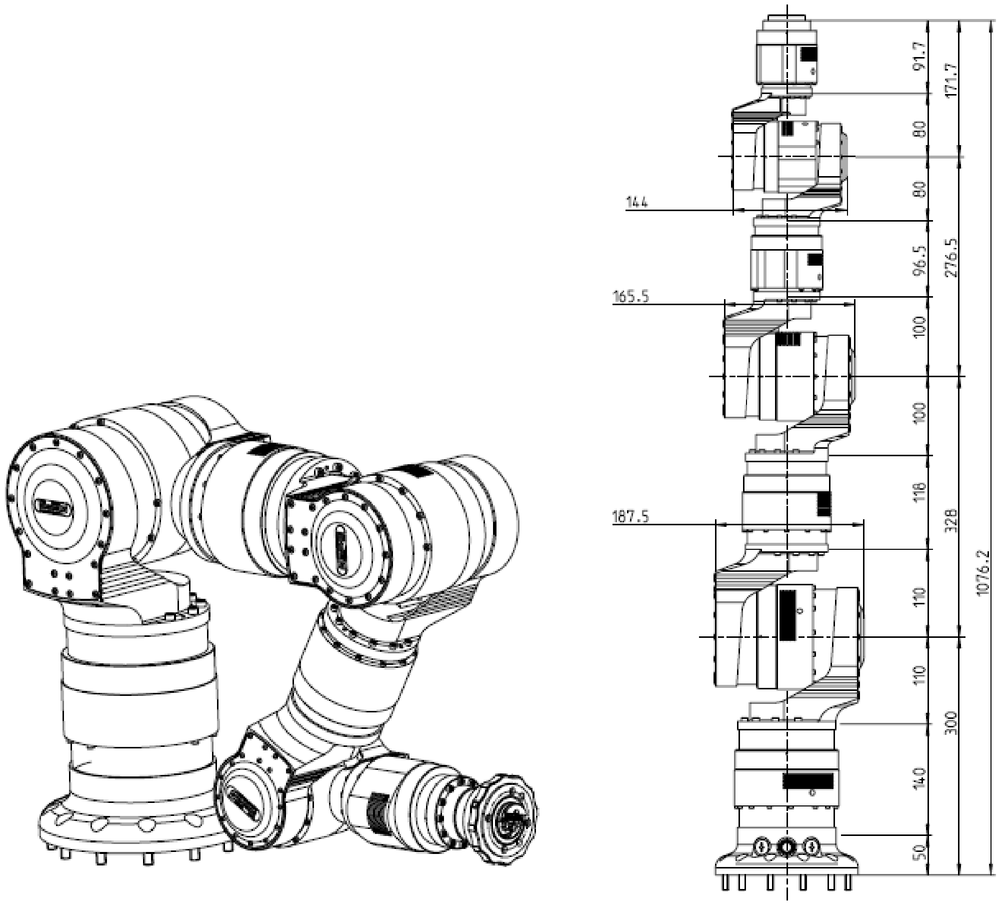


Fig. 1. Manipulator LWA3 with 7 degrees of freedom from Schunk GmbH, Germany, which is considered in the current project. [**5**]

# 2. Literature survey

Planning collision-free motions for robots from an initial to a goal position in static and dynamic environments is a fundamental robotic task. Usually the goal position as well as location and dimensions of obstacles are defined in low-dimensional *operational space*, whereas a feasible path has to provide a complete specification of the location of every point on the robot geometry or, equally, a trajectory of a robot in *configuration space*. This space represents the set of all transformations that can be applied to a robot given its kinematics. A robot with complex geometric shape is mapped to a point in configuration space. The main difficulty of motion planning arises from the fact that dimension of configuration space is equal to the number of degrees of freedom of a robot, which can be quite big for modern systems. This increases computational requirements for planning and motivates development of heuristic planning algorithms.

This literature survey gives an overview of methods used for motion planning and control. First, it focuses on classical approaches designed for static environments, including the sample-based and feedback ones, and then demonstrates, how these methods are expanded and augmented to handle dynamic environments. In the end, the detailed description of frameworks that this project focuses on is presented.

Part of this survey uses material from the term project completed last semester [**4**]. Namely, section 2.1 remains almost unchanged; sections 2.2 and 2.3 were augmented with a few more advanced approaches; section 2.4 was completely elaborated and description of only five planners was taken from last semester's report; section 2.5 was changed to be more relevant to the current project; section 2.6 is new.

## 2.1. Combinatorial roadmaps

The first attempts to solve motion planning problem aimed only for 2-dimensional configuration space where strictly deterministic approaches exploiting geometrical properties of the workspace can be applied. Such methods build *combinatorial roadmaps* that discretely and completely capture all information needed to perform planning [**6**]. One such approach is called *shortest-path roadmap* and it is illustrated with Fig. 2. The graph nodes are located at obstacle vertices and an edge exists in the roadmap if and only if a pair of vertices is mutually visible. Here feasible paths may actually touch obstacles, which must be considered during obstacle modelling.

Another technique called *vertical cell decomposition* is demonstrated in Fig. 3. In this case, the collision-free configuration space is decomposed into cells shaped as triangles or trapezoids. Roadmap vertices are created in the middle of each cell and in the centre of each boundary between the cells. A collision-free path can be easily computed with any graph-search algorithm. The cell decomposition is performed with “sectioning” the space with vertical planes.

The main advantage but also the main weakness of combinatorial techniques is their completeness, i.e. if a path exists, it will be found. Unfortunately, when applied to high-dimensional configuration spaces, the methods become highly ineffective due to combinatorial state explosion. However, some ideas first introduced in combinatorial methods may be a part of more complex algorithms.

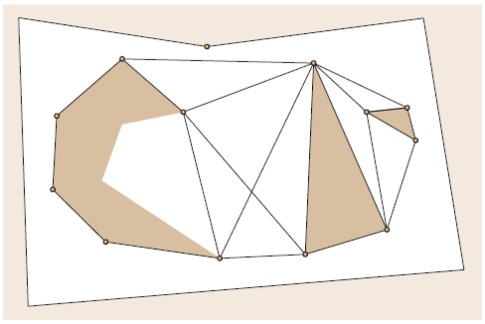


Fig. 2. Shortest-path roadmap example. [**6**]

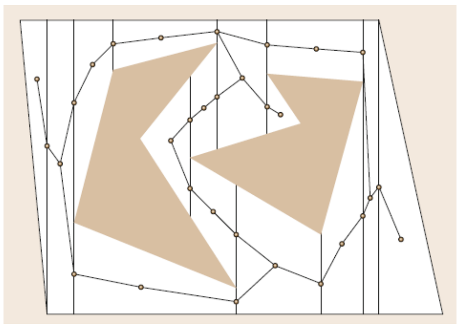


Fig. 3. Example of space vertical decomposition. [**6**]

### 2.2. Sample-based planning

Sample-based planning methods were designed in the 90s to overcome state explosion problems. Today these methods are by far the most common choice for industrial-grade problems. Sampling-based approaches usually achieve resolution completeness, meaning that they will find a solution if one exists, but may run forever if one does not, or probabilistic completeness, meaning that the probability tends to one that a solution is found if one exists (otherwise, it may still run forever) [**7**]. Sample-based planners can be classified in two types: multi-query and single-query planners. This section presents core functionality of both approaches.

*Multi-query planners* totally conform to their name: at the beginning a roadmap representing connectivity of configuration space is generated and later multiple path-search requests can be processed on its base. The roadmap is generated according to the following procedure:

- a sample configurations is chosen;
- if a robot with this configuration does not collide with obstacles it is added to a graph containing other collision-free configurations as nodes and paths between them as edges;
- a simple local planner tries to connect new placements to other nodes that are within some small region around the new one.

This algorithm is illustrated in Fig. 4. The process goes until connectivity of the configuration space is fully represented. To find a motion between a start configuration and a goal configuration, both are added to the graph and a path search across the graph is performed. This approach is called probabilistic roadmap (PRM).

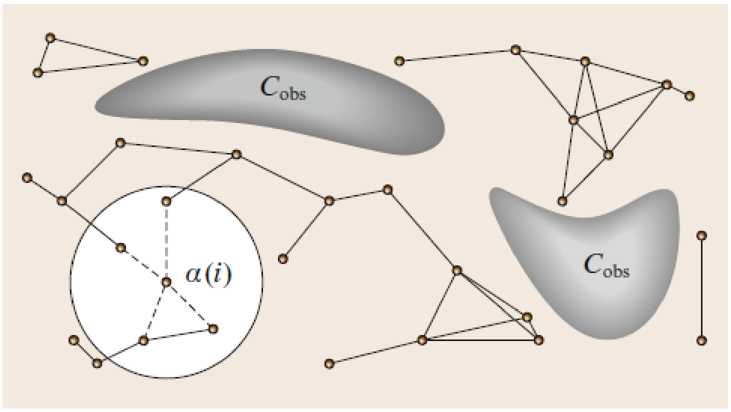


Fig. 4. Example of sampling-based roadmap construction. The process goes incrementally by attempting to connect each new sample (*α(i)*) to the vertices in the roadmap within a certain distance from the new sample. [**6**]

The main difference between the methods is in how new sample configurations are chosen. The first version of PRM ([**8**]) used random selection of samples which, of course, was not very effective because a lot of samples did not introduce new information about connectivity of configuration space. To reduce the number of samples and at the same time to improve roadmap quality other approaches were suggested. For example, it was proposed in [**9**] to put samples close to obstacles, which is similar to shortest-path roadmaps (see chapter 2.1). An approach given in [**10**], on the contrary, suggests to place samples as far from all obstacles as possible, so the probability of collisions is minimized. A more deterministic sampling technique, when samples can be placed only into vertices of a predefined grid, is described in [**11**]. A comparison of several multi-query methods is presented in [**12**]. It is shown that the best performance was demonstrated by a method combining probabilistic and deterministic approaches for sample-placement. Other multi-query methods suggest decomposing the general planning problem into low-dimensional workspace problem and configuration space problem. This allows to define obstacle-free regions in the space, which do not require many sample points to represent connectivity within them (Fig. 5). These measures reduce the number of samples and therefore lower the computational cost of the planning process.

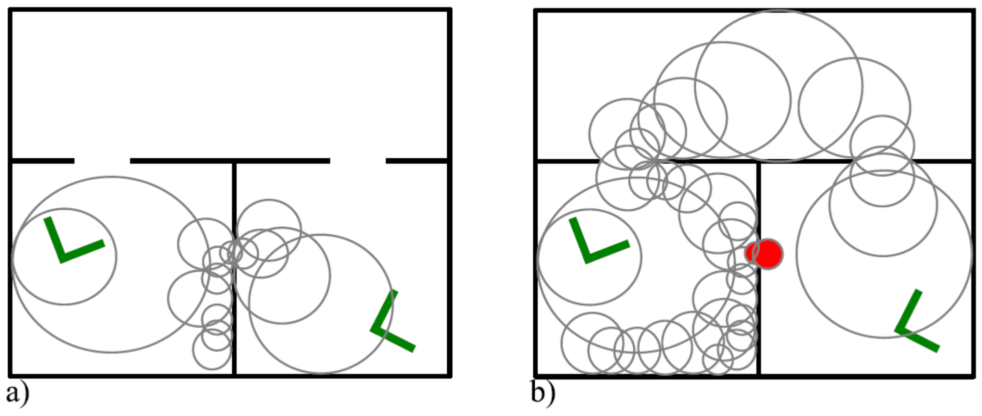


Fig. 5. Example of exploiting information about workspace for planning in configuration space. Each sphere represents a collision-free subspace and only few samples should be placed in each sphere to describe its connectivity. [**13**]

Sample-based methods of another type, *single-query planners*, do not build roadmap representing the whole space but generate a new tree-like graph every time they search for a solution. Targeting time efficiency, these graphs grow with the sole goal of connecting the initial and final configurations of one given problem. The method of *rapidly-exploring dense tree* (RDT) or *rapidly-exploring random tree* (RRT) originally presented in [**14**] demonstrates good performance and is widely used in various robotic applications. Pseudo-code algorithm of tree generation is represented in Fig. 6. After the initialisation (step 1) the algorithm works incrementally. In step 3 a random collision-free configuration is chosen and in step 4 an already existing graph vertex $q_{near}$ closest to this new sample is found. In step 5 NEW_CONF function selects a new configuration $q_{new}$ by moving from $q_{near}$ an incremental distance $\Delta q$ in the direction of $q_{rand}$. In steps 6 and 7 a new node and an edge are added to the graph. To ensure that the tree would finally converge to the goal, the goal configuration is chosen as $q_{rand}$ with a certain rate or probability. Fig. 7 illustrates how the algorithm explores the space.

Several variations of the method may be distinguished: unidirectional (with only one tree), bidirectional [**14**] (two trees, one growing from start configuration, another from goal configuration) and multidirectional [**15**](more than two trees). Usually a path found by PRM or RRT has a lot of unnecessary curves and requires smoothing.

Despite generally good results of RRT, authors of [**16**] claim, that the standard RRT method usually converges to a non-optimal solution under moderate practical constraints. To avoid this, a more advanced RRT* algorithm is suggested. In this version planning algorithm accounts not only for distance between vertices, but mainly for the cost of the path leading from the initial node to the new one. More specifically, the new vertex is connected not to the nearest vertex (as in line 7 in Fig.

6), but to the vertex, that incurs the minimum accumulated cost from the initial configuration up until $q_{new}$ and lies within some region close to the new node. RRT* also extends the new vertex to the vertices in this region in order to “rewire” the vertices that can be accessed through $q_{new}$ with smaller cost. It was proven that with these small, but crucial alterations the path found converges to an optimal solution and no path smoothing is required.

```
BUILD_RDT($q_{init}$, $G$, $\Delta q$)
1. $G$.init($q_{init}$);
2. for $k = 1$ to $K$
3.    $q_{rand} \leftarrow$ RAND_CONF();
4.    $q_{near} \leftarrow$ NEAREST_VERTEX($q_{rand}$,$G$);
5.    $q_{new} \leftarrow$NEW_CONF($q_{near}$, $\Delta q$);
6.    $G$.add_vertex($q_{new}$);
7.    $G$.add_edge($q_{near}$,$q_{new}$);
8. Return $G$
```

Fig. 6. Rapidly-exploring dense tree construction algorithm. Here G is the graph storing the configurations. Other notations are explained in the text.

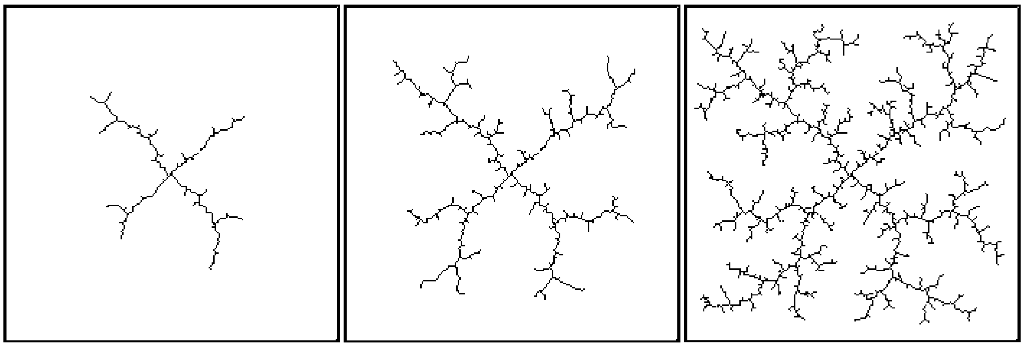


Fig. 7. RDT construction example. [17]

One interesting modification of RRT is presented in [18]. This method also builds a tree-like graph to explore configuration space of the robot, but a new node is chosen not randomly. Instead, some predefined motion primitives are used to form a new sample configuration, which is then assigned a cost value. The cost depends on closeness to obstacles and length of the path needed to reach this configuration from the initial one. The motion primitives can be of three types:

- When the end effector is far from its goal position, the motion primitive is just bending one or two joints by a predefined angle. At this time the main goal of motion is coming to the region close to the desired position of the end effector, therefore, the actuated joints are the ones close to the root and the step size is quite big (8° in case of one-joint motion).
- When the end effector is close to its goal position (within 10 cm region), a motion primitive based on inverse kinematics is calculated. It moves the end effector directly to goal position.
- When the end effector is at its goal position, then a motion primitive that drives it to a desired orientation is computed based on inverse kinematics.

As some of the motion primitives are not taken from predefined set, but are calculated directly during the planning process, the method is called *planning with adaptive motion primitives*. The path is constructed out of configurations with the lowest cost.

A different way to tackle dynamic constraints (including nonholonomic) during motion planning can be found in [**19**] and [**20**]. The methods are specially designed for systems with complex dynamics that are described by physical models instead of equations of motion. This is exploited during construction of the search tree. A new sample configuration $q_{rand}$ is not selected explicitly; instead, some control law (e.g. a vector of motor torques) is chosen randomly from a bounded region and applied to one of the nodes, already present in the tree. Then robot's forward dynamics is simulated in accordance with this control law until a collision occurs, a state-constraint is violated, or a maximum number of steps is exceeded. In the latter case a new vertex in the tree is formed.

However, usually integrating equations of dynamics of complex systems is computationally demanding. Therefore, the search should be guided to reduce the amount of calculation. Authors of [**19**] use deterministic approach to choose which vertex should be expanded. For this matter the whole configuration space is discretized into multilayer grid (Fig. 8,a), each cell of which is a polytope of fixed size. The cells are instantiated only when they are needed, so there is no need to store the whole grid structure all the time. A cell at every level stores its coverage rating. Additionally, the cells are classified as exterior if they are not completely surrounded by other already explored cells or interior otherwise. To ensure good space coverage the vertex to be expanded is more likely to be selected from a cell that is exterior and has low coverage rating (Fig. 8,b). Because of this mechanism the framework is called *Planning by Interior-Exterior Cell Exploration (KPIECE).*

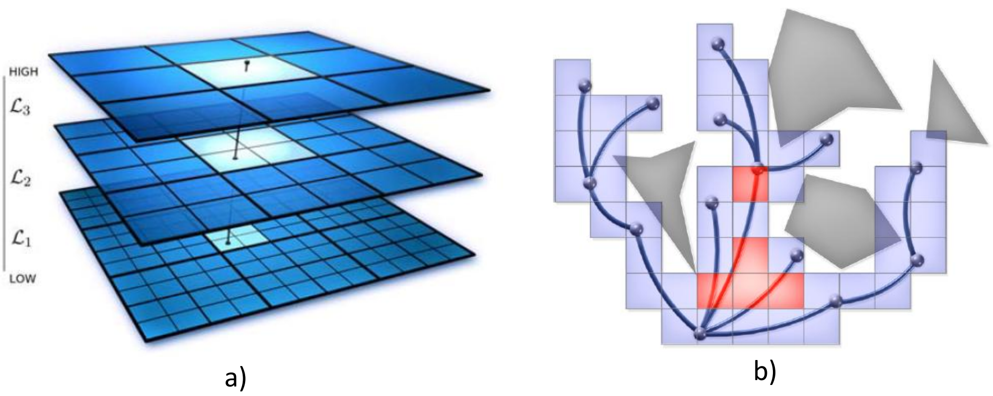


Fig. 8. Grid structure of KPIECE approach. a) Multilayer grids for space discretization. [**19**] b) Representation of a tree of motions and its corresponding discretization. Cells are distinguished into interior (red) and exterior (blue). [**21**]

Authors of [**20**] also use forward simulation of system dynamics for planning. However instead of forming the complex grid structures they exploit symbolic action planner (usually applied in artificial intelligence domain) to guide the search and to identify the regions of the space that planner should explore further.

## 2.3. Feedback planning

Feedback planning is one more approach that proved to be applicable in a vast variety of tasks. Feedback planners explicitly account for the fact that the information available during planning may be imprecise, that the environment may change during motion execution and that motion execution results in uncertainty about the state of the robot. Such planners produce not a specific path but construct a potential function defined in the permissible state space of the robot that determines appropriate motion commands. The robot motion is generated by moving in the direction where the potential function descends.

The classical *potential field method* (PFM) uses an analogy in which a robot is a particle that moves in the configuration space under the influence of a force field. The field has two components: attractive component that attracts the particle to the goal destination and a repulsive component pushing the particle away from obstacles (see Fig. 9). To apply the method to robots with several links Khatib proposed in [**22**] to calculate resulting forces for multiple points of the robot. Another extension of the method considers the current speed of the operating point and the maximum acceleration it can exercise. If the point moves fast, the region where the repulsive force acts is enlarged so there is enough space for the robot to slow down and avoid collision.

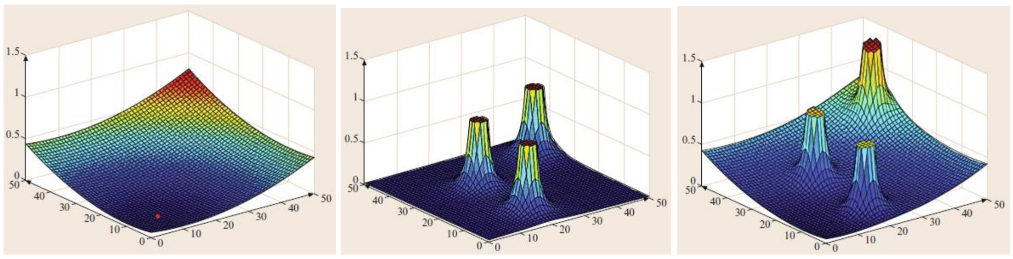


Fig. 9. Attractive, repulsive and resulting potential functions. [**6**]

Simple implementation of PFM has a set of problems inherent to the very nature of the method. In [**23**] the following problems are mentioned:

- Trap situations due to local minima (when obstacles form a dead end and the goal is behind them);
- No passage between closely spaced obstacles;
- Oscillations in the presence of obstacles (if a goal is behind an obstacle the robot, while moving along it, tries to go in that direction but is pushed back by the repulsive force; when the robot leaves the region of action of the repulsive force, it turns towards the obstacle again and the process repeats);
- Oscillations in narrow passages (the same effect as in previous case but with even greater oscillations).

To avoid these effects the same authors proposed to use vector field histogram (VFH) method [**24**]. The planning strategy has two stages. First is to calculate obstacle density in all directions in polar (or spherical) coordinates where the robot is in centre. The directions with low obstacle density become candidates for continuing movement. In the second stage one of the candidate directions is chosen to move towards the goal.

Another technique minimizing negative effects of simple PFM is the construction of a *navigation function* – a potential function (or a vector field) free of local minima ( [**25**]). Such function may have a specific global construction algorithm or be composed of several local potential functions (see Fig. 10 and Fig. 11).

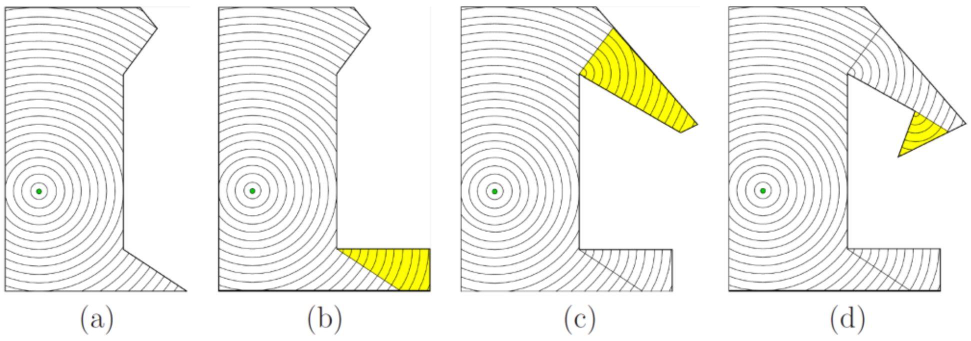


Fig. 10. Iterative construction of a navigation function using wavefront method. The point in the middle is the goal point. At the first step the function is constructed in the visibility region of the goal point. Then the corners of obstacles iteratively become intermediate waypoints and propagate the the global function to regions not visible from the goal point. [**25**]

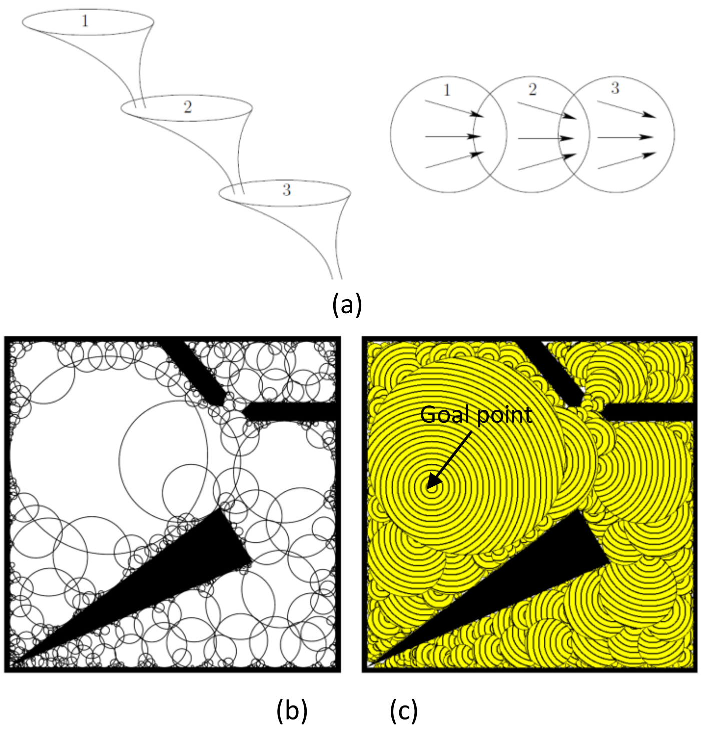


Fig. 11. Composing a navigation function out of multiple local potential functions. a) Each local function has a goal in the region of the next function in the cascade. The global goal is in the region of the last function in cascade. b) Configuration space is divided in regions with probabilistic sampling before forming the vector field. c) After the goal point is introduced, local potential functions are computed and united into a global navigation function. [**25**]

An interesting extension for potential-field planning was proposed in [**26**]. This approach allows to specify complex motion tasks like ``Visit area $\pi_2$, then area $\pi_3$, then area $\pi_4$ and, finally, return to and stay in region $\pi_1$ while avoiding areas $\pi_2$ and $\pi_3$" (Fig. 12). The properties of the regions, such as reachability, desired order etc. are coded with Linear Temporal Logic (LTL) language, which incorporates Boolean algebra with temporal information. Each area within workspace generates a potential field that corresponds to the complex goal. The fields can be switched during the task execution, e.g. area $\pi_2$ is attracting the robot at the beginning of the task and becomes repulsive in the end. Thus, the framework generates a hybrid LTL controller that is capable to drive a robot through a complex, multiple-staged task.

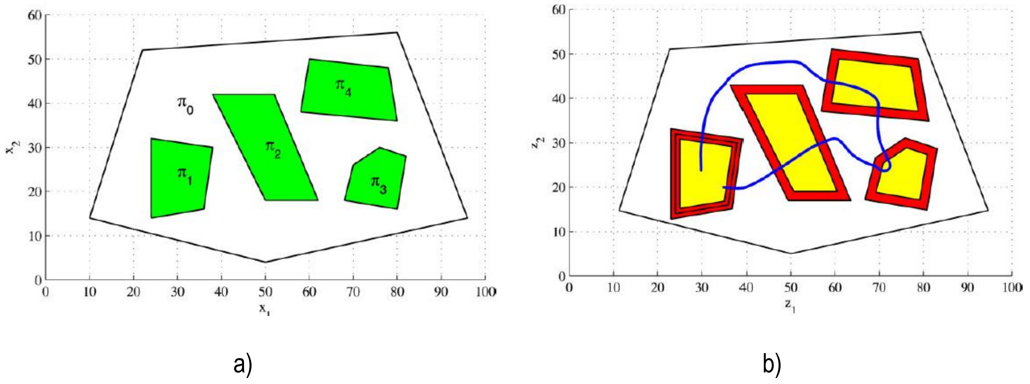


Fig. 12. Environment example (a) and the path found (b) for the framework, presented in [**26**].

## 2.4. Planning in dynamic environments

All methods described before have a common drawback - they are designed to work in static environments and are not suitable to handle moving obstacles. These methods compute data (a roadmap, a tree or a navigation function) about the whole configuration space at once and if an obstacle has moved the whole process must be repeated, which, of course, requires a lot of computation and may take inadmissibly long time, especially when computed in configuration space. Within a dynamic environment however, the system has the obligation to make a decision within a bounded time, otherwise it might be in danger by the sole fact of being passive. This limited available time for the system to make a decision, i.e. plan a motion, depends on the nature and dynamicity of the environment and is a hard real-time constraint. Therefore, special measures are required to tackle this complex task.

In this section the following groups of motion planners are presented:

- Motion planners for environments, where future locations of moving obstacles are known;
- Anytime planning;
- Combination of global planning with local replanning;
- Planning in velocity space
- Optimization-based planners;
- Path and trajectory deformation;

- Partial motion planning (PMP);
- Real-time adaptive motion planner (RAMP).

### 2.4.1. Motion planners for environments, where future locations of moving obstacles are known

Dynamic environments can be classified into two groups: when the future locations of moving obstacles are known and when they cannot be predicted. In the first case the two common approaches to plan movement are to add a time-dimension to the configuration space or to separate spatial and temporal planning problems (velocity tuning).

An example of the second technique is presented in [**27**]. It was developed for robot movement in such environments, as factory floors, where multiple robots are constrained to move along prespecified path networks (for instance, along lines painted on the floor). Thus, geometrically a robot moves along edges of precomputed or externally loaded roadmap, which allows avoidance of static obstacles. Moving obstacles, i.e. other robots moving along the same paths, are avoided only in time by tuning robot's velocities.

### 2.4.2. Anytime planning

Planning in absolutely unstructured and unspecified environment is much more challenging. Up-to-date there is no universal solution to this problem, and a variety of different approaches have been proposed. Many of them try to adapt RRT-search to dynamic environments, and it can be found in many complex motion planning environments.

Planning with RRT may take arbitrary time and thus the real-time constraint imposed by dynamic environment may be violated. Such situation can be avoided with *Anytime RRT** presented in [**28**]. The planner quickly finds a feasible, but not optimal (from time, energy consumption or smoothness point of view) solution and allows the robot start executing the path. While it is moving along the first piece of the path, the remainder is optimized according to data loaded at the beginning of planning or acquired during movement. Such strategy results in an almost optimal solution, as path execution often takes considerable amount of time.

Nowadays, there is already a bunch of planners exploiting this general idea, and the whole family is called *anytime planners*.

### 2.4.3. Combination of global planning with local replanning

Another concept that simplifies motion planning in dynamic environments is that usually only a part of a path is invalidated by an obstacle, and therefore only this part needs replanning, which can be done quickly.

An extension for a PRM-planner exploiting this concept is described in [**29**]. At the beginning a classical PRM-roadmap for static objects is created and stored. Then a moving object may invalidate some of its edges and vertices. When the roadmap receives a query, a path within static roadmap is calculated and analyzed. Depending on position of movable object several scenarios are possible:

- the obstacle has not invalidated edges in the path from the initial position to the goal. No special measures assumed;
- the obstacle invalidated some edges in the path. The planner tries to connect pieces of the path with a local rapidly-exploring tree. If the local "detour" of the obstacle is found it is incorporated into the path.
- the obstacle invalidated some edges in the path and the local method was unsuccessful. The global sampling (as during initial roadmap generation) starts to augment the roadmap with new vertices and edges.

Such scheme combining local and global planning allows maintaining roadmap in actual state. The idea of the method is illustrated in Fig. 13.

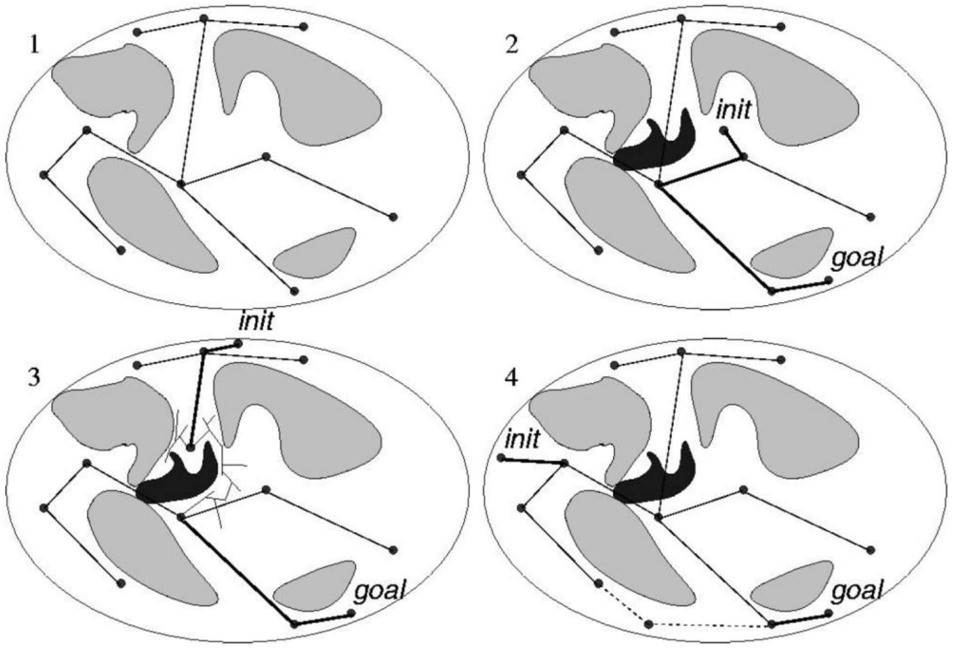


Fig. 13. Work of PRM-based planner for dynamic environment. A static roadmap is first computed in the configuration space of the robot (1). Processing the planning queries, a solution path can be found directly inside this roadmap (2) or via a RRT-like technique to reconnect edges broken by dynamic obstacles (3). If the existing roadmap does not permit to find a solution, new nodes are inserted and the roadmap is reinforced with new samples (4). [**29**]

A well-known approach adapting RRT to dynamic environment is called *lazy reconfiguration forest* (LRF) [**30**]. Forest-based planners maintain multiple trees instead of just a single tree. These planners plant several tree roots throughout configuration space and grow an RRT from each one until they are united into one tree. The LRF method first builds a tree to find an initial path. If an obstacle interferes with it, the interfered branch is not disregarded completely but is divided in several parts by the obstacle. One node from each of these parts becomes a root for a new tree and these trees start growing trying to reach each other so that the path could be reconstructed (see Fig. 14).

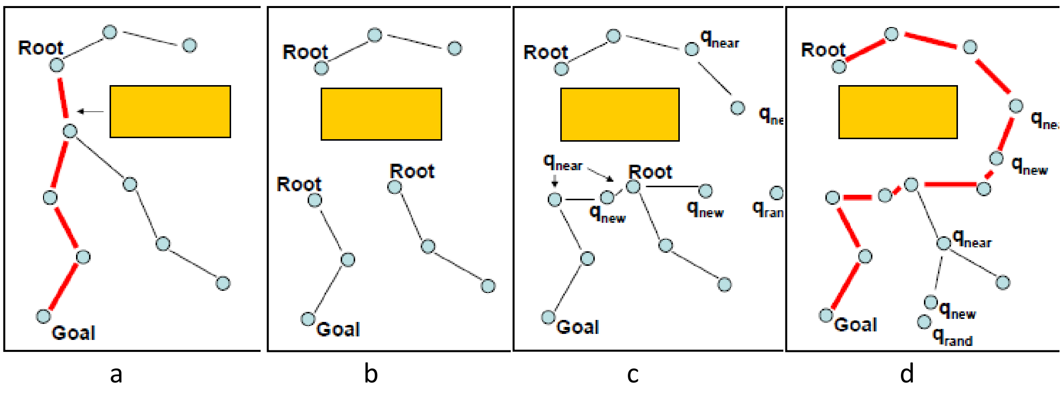


Fig. 14. Lazy Reconfiguration Forest working principle. An obstacle moves left toward the initial tree (a). Portions of the tree become invalid due to the obstacle. These portions are removed and the sub-trees not invalidated by the obstacle become new trees in the forest (b). The structures grow incrementally toward a sample configuration; two trees are merged, eliminating one root (c). The forest structure is united to form a path (d). [**30**]

An interesting approach for 2D planning is presented in [**31**]. It combines many ideas presented before. A roadmap of static environment is generated or loaded. Then it is augmented with time dimension to handle known moving obstacles. The path search is performed in state-time space (see Fig. 15 and Fig. 16) allowing to avoid dynamic obstacles (waiting at the same state is also allowed). If the framework detects that the identified path is not collision-free anymore, it tries to replan but only within the precomputed roadmap. To minimize replanning efforts A*nytime D** search algorithm is used. It is a modification of A* graph search algorithm that, firstly, works as anytime method and, secondly, searches a solution not from the initial state, but from the goal one. In A* search the cost values of the roadmap vertices become out-of-date with robot movement as the initial state changes all the time. The latter property of D* algorithm allows to avoid renewal of cost values of roadmap vertices, since the goal point is static, and, thus, to reuse the data computed in previous iterations for path optimisation and replanning.

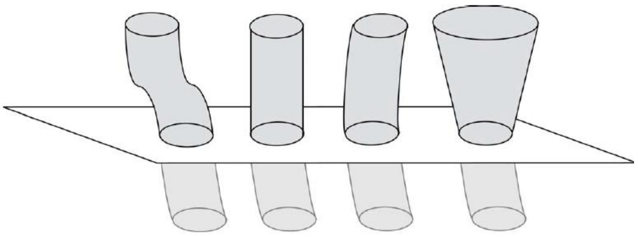
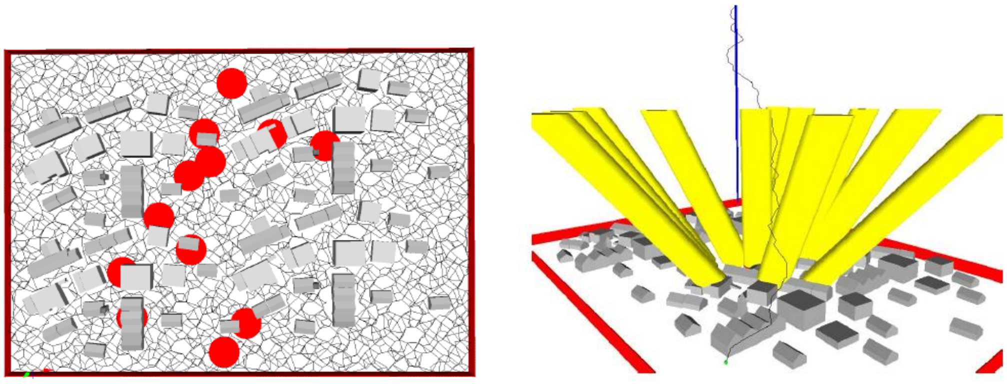


Fig. 15. Representation of trajectories of dynamic obstacles in state-time space. Vertical axis – time, horizontal plane – states. From left to right: a known trajectory; a trajectory of a static obstacle; an extrapolated trajectory based on previous motion; a worst-case trajectory based on current position and maximum velocity. [**31**]



a) b)

Fig. 16. a) A precomputed roadmap with dynamic obstacles (red). b) A path in state-time space (black). Blue is the goal configuration, yellow - extrapolated trajectories of the dynamic obstacles. [**31**]

### 2.4.4. Planning in velocity space

One more method of handling moving obstacles is to use velocity space for planning. An obstacle with known dimensions and velocity can be represented in velocity space as some region (a cone, if it moves straight (Fig. 17)). The task of a planner in this case is to choose such velocity vector, so that it drives robot towards goal and its end is not located within obstacle regions (in this case a collision would occur sometime in the future). Choosing one vector most probably will not be enough to reach the goal. Therefore, frameworks presented in [**32**] and [**33**] iteratively build tree of velocity vectors until one of the branches reaches a goal region. Each iteration movement of the robot and the obstacles is simulated in velocity space, then a new vector is chosen. Such planners cannot guarantee, that a global navigation task will be fulfilled, and require a global planner, that would define subtasks, feasible for iterative planning. A strong side of these planners can be demonstrated by a fact, that they may solve a task of crossing a street with car traffic, which is likely to end in a collision when solved with methods accounting for spatial information only.

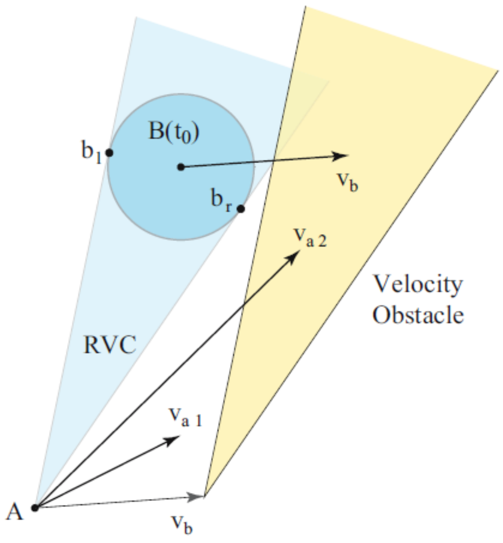


Fig. 17. A linear velocity obstacle. B($t_0$) – obstacle borders; blue cone – set of robot's velocities that would cause a collision if the obstacle was static; orange cone – set of robot's velocities that would cause a collision if the obstacle moved with the speed $v_b$; $v_{a1}$ – admissible robot's velocity vector, $v_{a2}$ – inadmissible robot's velocity vector. [**34**]

### 2.4.5. Optimization planners

There are also approaches that treat motion planning as an optimization task. One of them is CHOMP - *Covariant Hamiltonian Optimization for Motion Planning* [**35**]. It uses a two-stage procedure. First, a naive, probably not collision-free, initial trajectory from the initial position to the goal is created. During the second stage a modified version of gradient descent on the cost function is used to optimize the path with respect to being close to obstacles and path smoothness. It is one of few approaches that considers path's smoothness directly during planning and not while modifying previously found path. The way, the method handles dynamic obstacles is straightforward: it just continues optimization of already existing solution with respect to new data.

In [**36**] CHOMP is transformed into an anytime planner. It supplies robot with a suboptimal solution if the optimization task has not been solved within time budget and updates the trajectory during robot's movement.

### 2.4.6. Path and trajectory deformation

Another way to handle dynamic environment is not recomputing the paths, but deforming them when an obstacle approaches. This approach was first implemented in *elastic bands* [**37**]. The framework fully corresponds to its name as visually the path looks as an elastic band deformed by a moving obstacle. Each piece of the path is formed by two forces: an external force pushes it away from obstacles while internal force tries to make the path shorter. A similar approach applied to a nonholonomic mobile manipulator is described in [**38**]. The authors of [**39**] complement *elastic bands* with time dimension: now the trajectory can be deformed not only geometrically, but also in time. This allows, for example, to stop the robot for a while to let an obstacle pass by and then continue motion along the same path.

Even broader expansion of *elastic bands, Reactive Deforming Roadmaps*, is presented in [**40**]. Possible robot motions are described in a roadmap, each edge of which may be deformed by an approaching obstacle. Likewise, the roadmap vertices (also called dynamic milestones) may move if an obstacle comes too close (Fig. 18). In this way the whole roadmap reacts to changes in the environment and tries to adapt to them. The authors focus in their paper on planning for multiple robots in 2D space.

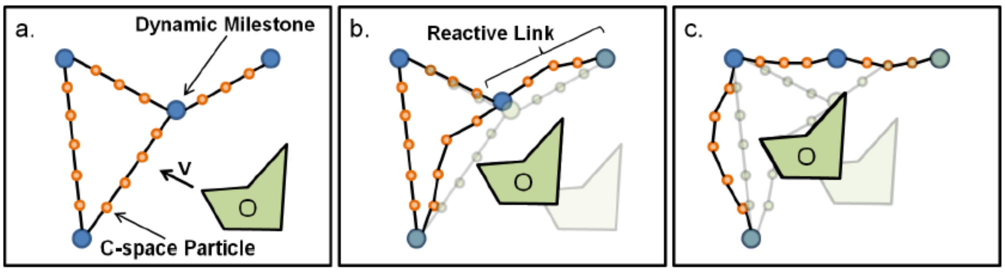


Fig. 18. Reactive Deforming Roadmaps (RDR). (a) The RDR contains a set of dynamic milestones and reactive links. (b) As the obstacle O moves, the dynamic milestones move as well and the reactive links of the roadmap deform to avoid the obstacle boundary. (c) If a path link deforms too much or is too close to the obstacle O, the link is removed. [**40**]

*Elastic strips* [**41**] is the first method from described before, that incorporates task-specific constraints important for realization of mobile manipulation tasks along with reactive obstacle avoidance. Thus, a robot tries to fulfil certain manipulation task (e.g. following a line with end effector or keeping upright orientation of the end effector) while avoiding moving obstacles (Fig. 19). In order to improve efficiency of the framework, most of the computation is performed in the workspace rather than in the configuration space.

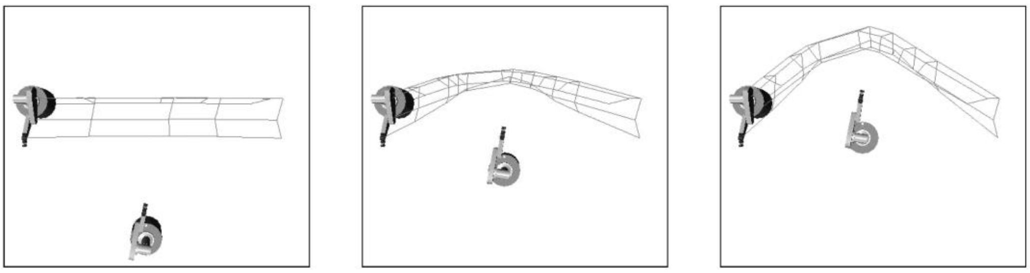


Fig. 19. Local path modification in presence of obstacles in *elastic strips* framework. [**41**]

A common drawback of path-deforming approaches is that at some point the changes in environment could affect the trajectory so intensively that local path modifications without global replanning may lead to invalid or arbitrarily bad motions.

### 2.4.7. Partial motion planner (PMP)

An interesting approach to satisfying hard real-time constraint during planning is presented in [**42**]. The authors of the paper argue that when obstacles move on their free will, as it happens in many real-life applications, their future behaviour is only partially predictable. Thus, it is likely that the model of the future that has been obtained will have limited validity duration. In such situation it is better to iterate a partial motion planning process taking as input a regularly updated predicted model of the future than to construct a global plan every time. From this assumption they develop a partial motion planner that works according to the following algorithm (Fig. 20):

1. An updated model of the future $B(t_i,\infty)$is acquired.
2. The state-time space is explored using a RRT rooted at the state $s(t_i+1)$ , where $t_i+1= t_i+\delta_c$.
3. At time $t_i+1$, the current iteration is over, the best safe partial trajectory $\phi_i$ in the tree is selected according to a given criterion and is fed to the robot that will execute it from now on. $\phi_i$ is defined over $[t_i+1, t_i+1 + \delta_{hi}]$ with $\delta_{hi}$ the trajectory duration.
4. Repeat the steps 1-3 every iteration until the goal is reached.

With this scheme a new best partial path is always generated within available time $\delta_c$.

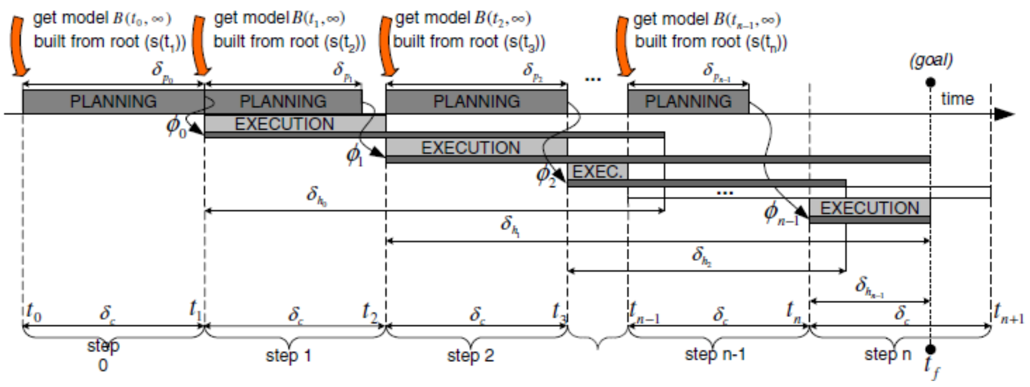


Fig. 20. Partial Motion Planning iterative cycle. [**42**]

Partial planning by its nature has to face safety issue: since PMP has no control over the duration of the partial trajectory is there a guarantee that the robot will never end up in a critical situations yielding an inevitable collision? To provide positive answer for this question, the authors introduce concept of *inevitable collision states*

(ICS) - the states for which no matter what the future control law applied to the system, a collision with the obstacle occurs. Thus, if a partial plan is ICS-free, it is ensured that it is safe. It is also shown that for a car-like robot computation of ICS can be simplified greatly.

### 2.4.8. Real-Time Adaptive Motion Planning (RAMP)

A common approach to increasing reliability of a system is redundancy when the most critical components of the system have a backup that can come into operation quickly in case of failure of the currently active instance. A similar idea is exploited in RAMP approach presented in [**43**]. If a path may be invalidated by an obstacle, it is reasonable to have several paths ready, so that the obstructed one can be immediately substituted. RAMP is designed for mobile manipulators (a manipulator on a mobile base) and provides reasonable results even in this high-dimensional configuration space.

The framework simultaneously maintains and updates multiple trajectories in state-time space going from the current robot position to the goal region. The exact goal configuration for each trajectory is randomly chosen from the goal region, which facilitates homotopic variety of the trajectories (Fig. 21). The trajectories are planned with an anytime planner, so most of them are not optimal and/or even infeasible at the beginning. According to the anytime planning concept, trajectory updating and optimization go simultaneously with robot movement. Optimization process has stochastic nature. Each iteration one of multiple possible operations is applied to one of the paths. The set of operation includes, but is not limited to adding or deleting a milestone configuration, substituting a milestone configuration with another, stopping the base or manipulator of the robot for a while etc. If the new trajectory has lower cost value (infeasibility is also a cost penalty), one of present trajectories of same homotopic class (not exactly one with the lowest cost) is substituted. The stochastic nature of trajectory optimization and maintenance allows to preserve trajectories of different nature and to be ready to big changes in the environment.

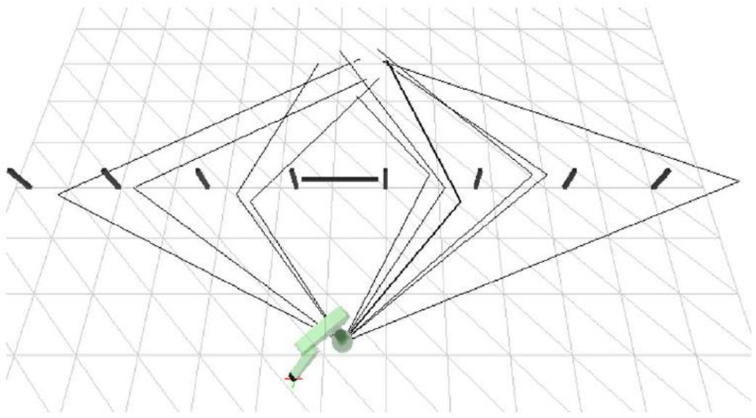


Fig. 21. A template task for RAMP. A population of trajectories is required as doors can close and open unexpectedly. The initial trajectory set has a good diversity to cover the environment. The trajectories going through the same gate belong to one homotopic class. The figure indicates base trajectories only. [**43**]

## 2.5. Elastic roadmaps

*Elastic roadmaps* is a state-of-art approach for planning within unstructured environment. It is a comprehensive approach for motion generation that combines together planning and control. It allows to perform mobile manipulation tasks in dynamic environments and to satisfy various constraints with different feedback frequency requirements. As a reference scenario for mobile manipulation the authors in [**44**] chose inspecting a pipe when a robot moves along it while keeping the end effector (with an inspecting tool installed) in proximity to the pipe and avoiding obstacles on the way. The whole framework combines sample-based and feedback planning and works according to the following algorithm.

1. A manipulation task is specified in operational space. As an example, it can be following a line with the end effector or keeping the end effector in proximity to a wall.
2. Obstacle-related milestones are created. A milestone is a virtual robot that has a specific task. The milestones are placed near features (a corner or the middle point of an edge; see Fig. 22) of obstacles and their primary task is to stay in proximity with the feature while trying to fulfil manipulation task specified in step 1 and avoiding obstacles. Thus, if an obstacle moves the corresponding milestones move along with it. The placement of milestones depends only on workspace information.

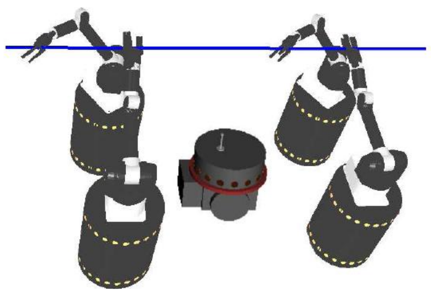


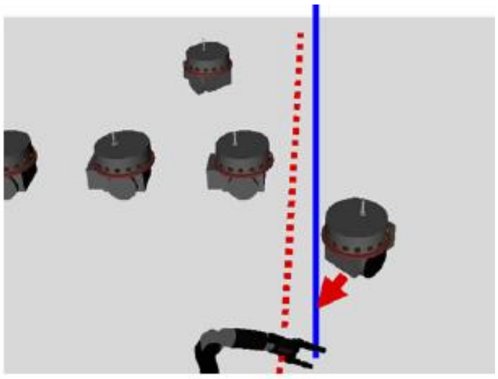
Fig. 22. Example of milestone placement in *elastic roadmaps* framework. [**44**]

3. A roadmap is generated. The roadmap contains the information about milestones and connectivity between them. Two milestones have an edge between them if their corresponding characteristic points satisfy visibility criterion (e.g. can be connected by a straight line that does not intersect the obstacles). Thus, the roadmap is a graph representing connectivity of the workspace.
4. A path is determined in the roadmap. The start and the goal configurations are added to roadmap as milestones and an appropriate path is found with a graph search algorithm. The vertices of the graph (milestones) contain information whether a milestone is able to perform the manipulation task from step 1. It is assumed that if two milestones have an edge between them and both satisfy task constraints, movement from one to another is possible without violation of the task. The path is computed only through milestones fulfilling the manipulation task. Therefore, manipulation task is accomplished along the whole path.
5. A robot moves along the path. As the path is specified only in the workspace, a navigation function with the sink at the next milestone in the path is formed. When the milestone is reached, the new navigation function leading to the next milestone is computed. The robot's motion along the navigation function is controlled by a task-level controller which allows performing of several prioritized tasks at the same time.

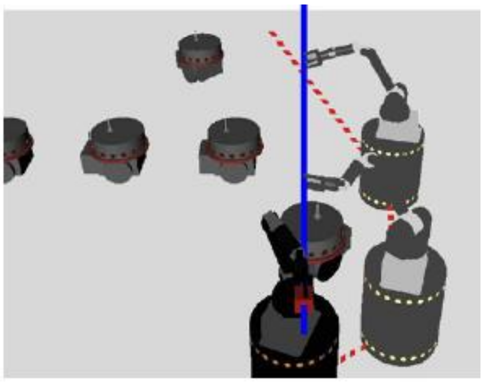
Thus, *elastic roadmaps* form a hybrid controller switching between navigation functions and a trajectory in configuration space is never computed explicitly. This allows updating the path in dynamic environment with a rate sufficient for mobile manipulation tasks. An example of framework's functionality is presented in the Fig. 23. To reduce the amount of computation milestone placement, checking connectivity between them, path generation and modification is performed strictly in operational space. The transition from operational to configuration space is done separately for each milestone via task-level controllers. The task-level controller is one of the key components of the current project and its functionality is presented in details in section 3.4.

The framework provides generality in task constraints specification but it is not complete, i.e. it may propose an infeasible path or fail following a feasible path. The authors of the framework realize that and introduce failure detection mechanism to increase system's completeness. They identify three types of failures. Failures of the first type appear when the robot cannot find a motion between two milestones connected in the roadmap. It can be a consequence of a mistake in connectivity controller, as it only checks visibility between some points of two milestones but does not takes in account geometric dimensions of the links. In this case, the framework should detect that no progress has been done for some time, invalidate the current edge between milestones and find a new path. Failures of the second type correspond to violation of manipulation task: when a robot or any of milestones in the path are not task-consistent any more, the framework tries to find a new path. If the new task-consistent path has not been found the robot should recover as soon as possible to the task-consistent configuration or to the closest task-consistent milestone. If these recovery strategies still do not allow finding new path, a failure of the third type is generated. It corresponds to incompleteness of the method and, in this case, the task cannot be solved with *elastic roadmaps*.

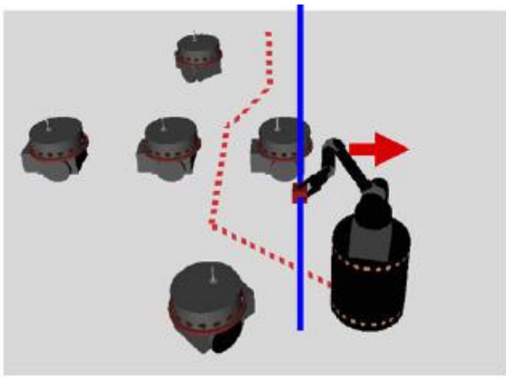
There are several factors that limit application of the framework. The first is its completeness. It is caused by the core algorithm of the system: place milestones wisely and hope, that the task-level controller finds a feasible path between them. The failure detection mechanism described in the previous paragraph tries to minimize this drawback but it is impossible to eliminate it completely. The second limitation is that the obstacles in the framework have to be always represented with their bounding boxes, which is essential for milestone placement. Such representation may conceal shape and structure of compound obstacles and result in a suboptimal path or inability of the framework to solve certain tasks. If we divide an obstacle into multiple objects for better representation of its shape, the number of obstacle-related milestones increases greatly which in turn leads to a boost of computational requirements since each milestone is controlled with a complex task-level controller. One the more factor that may also affect performance of the system is visibility criterion, as it considers only some reference points of the robot and not its body in whole.



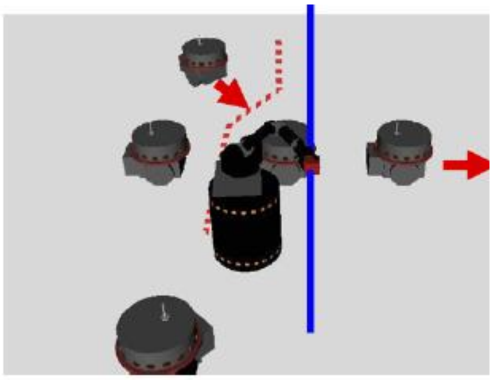
(a) The initial desired motion is indicated by the red dashes. It directly connects the current position of the actual robot to the milestone at the goal location. The robot on the right starts moving into robot's path, as indicated by the arrow.



(b) A new motion is selected from the elastic roadmap. It goes through two more milestones, indicated by the transparent robots and circumnavigates the moving obstacle.



(c) Due to the motion of the three robots (indicated by the arrow), a new motion is shown. It goes through three milestones of the roadmap (not shown) before reaching the goal position



(d) As the three robots continue their motion, another robot starts to move (again indicated by the arrows). Yet another motion is selected from the elastic roadmap.

Fig. 23. Example of *elastic roadmaps* functionality. The robot performs a task that requires the end-effector to traverse a line in space. Multiple moving obstacles obstruct robot's path. *Elastic roadmaps* generate collision-free and task-consistent motion. [**44**]

### 2.6. Dynamic roadmaps (DRM)

In this section we discuss a framework for real-time path planning originally presented by Leven and Hutchinson in [**45**] and later refined by Kunz et al. in [**46**]. This approach is based on PRM method but it introduces a number of improvements that aim at handling unstructured dynamic environments. The key idea exploited is that the cost of planning can be divided over many planning episodes. This provides a justification for spending extensive amounts of time during a preprocessing stage, provided the resulting representation can be used to generate plans very quickly during a query stage. Therefore, the method has two stages: offline precomputation and online path search.

It is assumed during offline precomputation that no obstacles are present in the robot’s workspace. Two tasks are performed in this stage. First, a probabilistic roadmap is generated in configuration space. After the roadmap is formed, it is mapped to 3D workspace. To do this the workspace is divided in cubic cells within a grid with constant step size. Each node (configuration) and edge of the roadmap are examined to define which workspace grid cells it intersects with (Fig. 24). This information is then stored in the workspace cells. Thus, each cell contains a list of roadmap nodes and edges that are in collision with the cell.

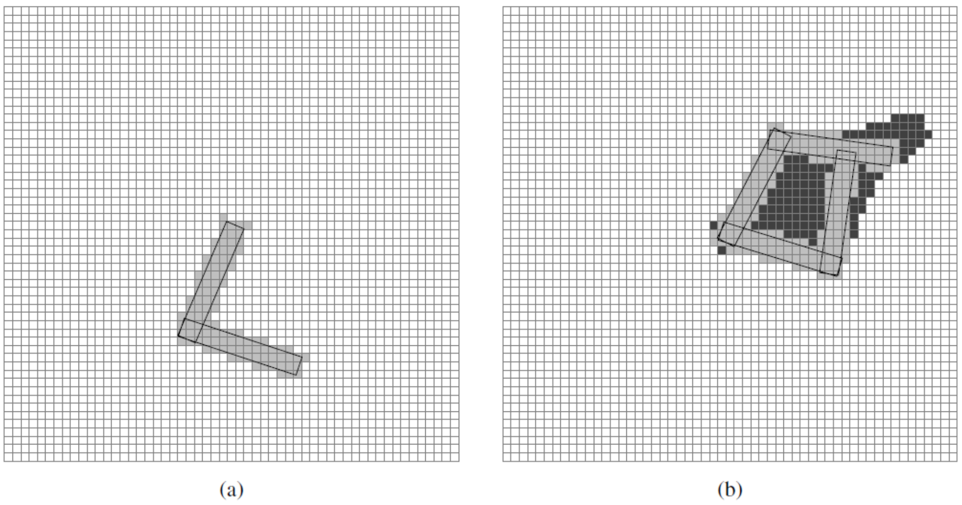


Fig. 24. Mapping a configuration and an edge to workspace for a 2-DOF planar robot. [**45**]

The data stored by the cells is then used during online planning. When an obstacle comes into robot’s workspace, it occupies some of its cells. All nodes and edges colliding with these cells become invalid, as they would also collide with the obstacle. Thus, these parts of the graph are removed from the roadmap and are not considered during path search.

Since there is no time limit on precomputation part mapping of configurations and edges may take arbitrary long time. Kunz et al. use a brute approach to map a configuration. A big number of points is uniformly distributed within the robot’s body and for each of them a cell containing the point is defined. The method, used by Leven and Hutchinson is more accurate: they directly compute a set of grid cells that represents a polyhedron. The latter method is quicker and more precise, but harder to implement. To map an edge, the configuration corresponding to its midpoint is voxelized first. Then the edge is recursively subdivided in two parts and the process repeated until no new occupied cells are added anymore. The cells that are occupied by the two endpoint configurations of the path segment are not considered for the set of cells occupied by the path segment.

The framework works in configuration space only, so the authors have to pay big attention to such issues as selection of sampling techniques, so that whole configuration space is sampled, and choosing appropriate distance metric to define which nodes should be connected with edges. Another matter that comes into play is size of the roadmap and cells' data. In Kunz's implementation several hundred megabytes was needed to store the data for 7-DOF manipulator and it took approximately 20 hours to finish the offline stage. Most of the time was spent on collision checks of roadmap edges.

On other hand, performance of the online planner benefits significantly from using the precomputed data. It can compute an obstacle-free path within hundreds milliseconds and is capable to handle moving obstacles. The research presented in [**47**] compared performance of DRM and RRT. Three testing scenarios were studied: a planar hand with 4 DOF, a planar two-handed robot with 7 DOF and a Robonaut humanoid model with 17 DOF. The test results demonstrate, that in the first two scenarios planning with DRM was much faster (more than 4 times faster in 4 out of 5 tests), than RRT. Moreover, for the robot with 7 DOF DRM was able to solve more tasks, than RRT. However, for the Robonaut scenario the approach was not so effective and quite big part of the tasks was not solved. This can be explained by the assumption, that the used roadmaps were not capable of adequately covering the free configuration space with such big number of dimensions. One more observation made by the researchers is that DRM is effective in finding path to extreme postures as they store complete data about environments with complex configurations.

The current project originates from a semester project that focused on mobile manipulation ( [**4**]). In this context it is worth mentioning a method that expands DRM into this domain. The main challenge of using DRM for mobile manipulation is increased and potentially unlimited configuration space. In this case, it is impossible to describe it with sampling. The authors of [**48**] suggest a multilayer structure. The top-level planner is a simple RRT algorithm that finds new collision-free nodes in configuration space and determines the global path. The nodes of the graph serve as subgoals for local DRM-planner that may quickly react to moving obstacles and find the actual detailed path from one subgoal to another in the configuration space. In this case, the top-level planner does not need to check edges for collisions, since it is done on the lower level.

In conclusion, it would be reasonable to compare the frameworks that are the key components of the current project. *Elastic roadmap* is a universal and powerful approach that allows handling of dynamic environments and preserving task-constraints throughout the path execution. It mostly works within operational space and is applicable to robots with various structures, including mobile ones. However, the very nature of the framework makes it incomplete and it may provide non-optimal solutions in cluttered environments with complex-shaped obstacles. DRM, on the contrary, is probabilistically complete and demonstrates good performance in situations, when extreme configurations should be used. On the other hand, it does not consider task constraints and can be applied only to robots with limited workspace.

## 2.7. Summary

In the survey an overview of various methods for motion planning and control was given. At the beginning classical methods, such as combinatorial roadmaps, were presented. Then basic algorithms and their more advanced versions for sample-based and feedback planning in static environments were explained in details. After that, a variety of methods applied to planning in dynamic environments was reviewed.

In the end, the key components of the current project were discussed. It was shown that *elastic roadmaps* framework combines features of sampling-based and feedback motion planning and uses both global replanning and local path modification to maintain a task-consistent path in dynamic environments. It was also demonstrated that dynamic roadmaps (DRM) allow quick planning even in environment with moving obstacles due to a special mapping technique. DRM does not consider task constraints and is only capable to find an obstacle-free path from one configuration to another.

# 3. Design

This chapter explains all basic features of the proposed framework. First, a general concept of the suggested solution is explained; then the details about design of offline preprocessor and graph search algorithm are presented; section 3.4 describes functionality of the task-level controller and computation algorithms for some of its secondary elements; in section 3.5 we overview the task hierarchy for the current implementation of the task-level controller and design of each task; section 3.6 gives description of simple simulation model used to transform computed generalized forces to a new configuration.

The general structure of the task-level controller and some of its tasks were designed during the specialization project last semester. The description of its functionality and implementation details (section 3.4) were taken from the report to that project ( [**4**]) and adapted to be more relevant to the current framework. Section 3.5 mostly refers only to the current project.

## 3.1. General description of the proposed framework

As already stated above, the aim of the current project is to design a motion planner that would be capable to handle dynamic environments and to satisfy task constraints during robot movement. For this goal we combine features of *elastic roadmaps* and DRM in one framework.

In most cases end user of a robot would specify a task in operational space; therefore constraints imposed by the task are also specified in operational space. Talking about task constraints, we first of all mean that position and/or orientation of the end effector of manipulator cannot change arbitrary during its motion. Here are some examples of tasks and corresponding task constraints: carrying a glass of water while keeping it upright and above a certain level above the ground; painting something on a wall with a sprayer while preserving a certain distance from the wall and aiming with sprayer at it; pipe inspection with a videocamera when the pipe should always stay within camera’s view and, possibly, at a certain distance to be always in focus.

Like DRM, the proposed framework has two stages: offline preprocessing and online planning. In the offline stage a roadmap covering robot’s workspace is generated and mapped to workspace cells with technique, similar to one used in DRM.

The main difference at the offline stage is that the roadmap is generated not in configuration, but in operational space. The workspace is discretized with a 3D grid. Each node of the grid represents a possible position of the end effector and stores information about six configurations. All configurations refer to the same position of the end effector, but the corresponding orientations are different. The six orientations coincide with the positive and negative directions of the axes of the inertial frame (i.e. correspond to the “up”, “down”, “right”, “left”, “forward” and “backward” directions when viewed from the inertial coordinate frame). In fact, for almost every node some of orientations are infeasible. In this case a flag that the corresponding configuration does not exist is stored in the node. In such manner we, firstly, identify, which positions the end effector may occupy, and, secondly, represent roughly, which orientations are feasible at these positions. The nodes that have at least one feasible configuration are included into the roadmap. An edge in the graph exists between any two adjacent nodes.

Apart from forming a grid for the end-effector position, the workspace is discretized one more time into cells that are used for mapping configurations to the operational space. This mechanism is absolutely identical to the one used in DRM. After mapping each workspace cell stores which configurations (stored in roadmap nodes) it collides with.

At the end of the offline stage the roadmap and the data structure aggregating the cells are stored on the hard drive.

The online planning stage consists of two processes that run simultaneously. The first one handles obstacle movement and performs graph search. When a new obstacle arrives or already identified obstacle moves, it is mapped to the cells of the workspace, and all configurations that collide with these cells are invalidated according to the precomputed data.

The graph search establishes the actual path from the initial node, which is the closest node to the current position of the end effector, to the goal one. During the search each node is tested for task consistency. If position of the end effector is constrained, the position of the node is checked. If the task constraint is applied to the orientation of the end effector, first the desired orientation vector is computed for the node. Then we determine, the projections on which axes of the inertial coordinate frame are dominant for this vector, and check if the corresponding configurations exist within the node and if it is valid (i.e. collision-free). Thus we approximately verify whether the node is task-consistent or not. Only nodes with valid task-consistent configurations can be included in the path. The result of the graph search is a sequence of nodes’ positions that lead to the goal point. With a dense enough grid of nodes we can ensure that the found collision-free path can be followed by the end effector in approximately task-consistent manner.

The other process running online is the task level controller. It is a structure that actually controls the robot in accordance with a set of prioritized tasks (they are also called *behaviours* in this report) that act in the nullspaces of each other. The set includes several safety behaviours, e.g. avoiding joint limits and avoiding obstacles, a behaviour controlling robot's end effector orientation and a behaviour that moves the end effector towards the goal position specified by the planner. The structure of the task-level controller ensures, that orientation constraints will be preserved during the path execution if the safety behaviours allow. The planner supplies the path-following behaviour with a new goal as soon as the previous one was reached.

To sum up, by exploiting the data obtained during offline stage the planner is able to quickly find a path of the end effector in the operational space that can be followed in obstacle-free and approximately task-consistent manner. During the path execution the task-level controller tries to exactly satisfy the task constraints and tries to follow the found path while avoiding collisions with obstacles and inadmissible configurations. The details of the framework and main design decisions are discussed in the following sections.

## 3.2. Offline preprocessing

### 3.2.1. Roadmap generation

Operational space of a manipulator has six dimensions: three coordinates of the position of the end effector and three angles representing its orientation. In the offline stage we discretize the operational space to form the roadmap, identify which regions of the space (both in position and orientation) are reachable and then use this information during motion planning and control.

As position and orientation dimensions of the operational space have different nature, they need to be treated separately. To discretize the position dimensions, a 3D grid is formed, each vertex of which corresponds to a possible position of the end effector. In our project the grid has fixed step size, but there no obvious restrictions to use grid with variable node density.

A space of orientations of an object with fixed position is a spatial angle of $4\pi$ (in this case we assume, that orientation is considered only as direction of Z-axis of end effector's coordinate frame; directions of corresponding X and Y axes are not important as they may be easily adjusted by the last revolute joint of the manipulator when the Z-axis is oriented as desired (Fig. 25)). The space is approximated with six directions: up, down, forward, backward, left and right when observed from the inertial coordinate frame. Thus, we can say, that in preprocessing stage the orientation space of each node of position grid is discretized into six states. It will be shown in the section 3.3.2 that more states are used during the graph search and planning and the orientation space is not discretized at all during path execution.

Thereby we can represent the whole 6D operational space of a robot as a grid, corresponding to the positions of the end effector, where each node of the grid contains a set of possible orientations of the end effector. Thus, a grid node may potentially contain 6 points of the six-dimensional operational space. In reality for the vast majority of the nodes some of these points are not reachable due to kinematic limitations. This is checked by solving an inverse kinematics problem.

The grid described above forms a graph that is used as roadmap in planning. The edges in the graph exist between all neighbouring nodes that contain at least one valid configuration since they are close to each other in operational space. All edges have the same weights equal to 1.

### 3.2.2. Grid resolution choice

One of the factors that affect grid resolution is the fact that we do not check the edges between the nodes for collisions. It does not make any sense really, as each node contains six configurations and it is impossible to predict offline, connections between which of them are going to be used during motion.

However, we would like to guarantee at least up to a certain degree that there are no obstacles between two adjacent nodes. For this matter we consider two configurations that are contained in the neighbouring nodes and correspond to the same orientation of the end effector. If there is no gap between the two configurations, it is most likely that no obstacle will obstruct the direct path between them and thus we increase the probability that the task-level controller is able to perform such movement basing only information about the state of the end effector. Thus the grid step size should be less than geometrical dimensions of the end effector since it is usually the smallest part of manipulator and it covers the biggest distance while moving from one node to another. This gives the upper bound for the grid resolution.

The lower bound should be defined from task accuracy specifications. It is also obvious, that the smaller the resolution is, the bigger the roadmap graph (and also files, storing the roadmap and it's mapping to the workspace) is.

In the current project the diameter of manipulator's end effector is 7,5 cm. The grid resolution was chosen to be 5 cm for the roadmap and 3cm for workspace cells.

### 3.2.3. Inverse kinematics solver (IKS)

The IKS play a great role in the framework. Firstly, it defines which positions and orientations are achievable by the robot. Secondly, the results produced by IKS are used in mapping configurations to workspace.

The algorithm of IKS assumes making a very important design decision related to redundancy resolution. In this project we work with 7-DOF manipulator from Schunk GmbH, Germany (Fig. 1 and Fig. 25). As the operational space may at most have 6 dimensions, the manipulator is redundant and some measures need to be assumed.

Usually redundancy is solved by specification of a secondary task for the robot. Such task can have a goal of avoiding singular configurations and joint limits, optimizing joint torques or providing decoupled force/position control of the robot. In the current implementation the secondary task is to provide, that configurations corresponding to neighbouring nodes are also close in configuration space. This is extremely important, as during the planning stage we do not form a full path in configuration space, but specify some waypoints that should be followed by the end effector. Thus, if two configurations corresponding to the neighbouring waypoints are far away from each other in configuration space, the assumption, that the task-level controller may solve the task of waypoint following, is likely to fail. It is also important to notice, that joints 1, 3, 5 and 7 of the manipulator that is considered in the project have limits more than π in both direction. Because of that the situations when a robot has to rotate 360° to get to the adjacent node can be excluded from our consideration during offline preprocessing.

In our project we define the secondary inverse kinematics task as keeping the angle value of the joint 3 at zero and assigning only positive angle values to the joint 4. The first rule transforms our redundant robot into classical 6-DOF PUMA-like manipulator, for which inverse kinematics may be easily solved. The second rule may be justified by the assumption that the most obstacles do not fly in the air, but stand on the ground. With the constraints specified above we explore configurations that approach obstacles "from the top" for which the probability to go around the obstacles standing on the ground is high. These kinematics restrictions are valid only during offline stage. The robot may use all its DOF during actual moving.

Imposing the constraints on two of the robot's joints limits the part of configuration space that is explored during preprocessing stage. This is an unavoidable drawback of the proposed framework and it will be discussed more in section 6.

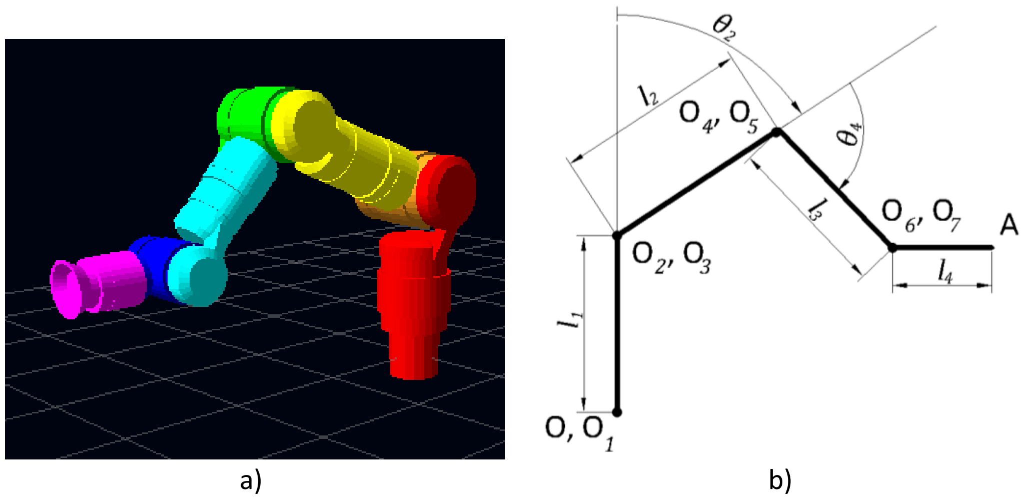


Fig. 25. The model of the robot (a) and its approximation for inverse kinematics solver (b). The point $O$ marks the origin of the inertial frame. The points $O_1 .. O_7$ denote origins of coordinate frames attached to robot links. The rotation axes of the joints go through these points and coincide with main axes of cylindrical links of the model in the figure (a), i.e $Z_1$ corresponds to the red link, $Z_2$ to the orange link, $Z_3$– to the yellow one etc. The axis $Z_7$ coincides with the main axis of the violet link and represents orientation of the end effector.

With the assumptions presented before solving inverse kinematics becomes an easy task. The robot now can be modelled as 4 links, 3 of which are connected with revolute joints, and the last one (the end effector) is attached with a spherical joint (Fig. 25). The input for an inverse kinematics task is the position of the end effector (point $A(x_A, y_A, z_A)$) and its orientation as a unit vector of Z-axis of end-effector's coordinate frame $\overrightarrow{Z_7}$. From these we can easily find the desired coordinates of the point $O_6$:

$$\overrightarrow{OO_6} = \overrightarrow{OA} - \overrightarrow{Z_7} \cdot l_4.$$

Now we should check whether this point is reachable. If $\left|\overrightarrow{O_2O_6}\right| > (l_2 + l_3)$, then the point is too far and no solution exist. In the opposite case by applying cosine theorem to the $O_2O_4O_6$ triangle we can obtain rotation angles of the 1st, 2nd and 4th joints:

$$\theta_1 = atan2(y_{O_6}, x_{O_6})$$

$$\theta_4 = \pi - \arccos\left(\frac{{l_2}^2 + {l_3}^2 - \left|\overrightarrow{O_2O_6}\right|^2}{2 \cdot l_2 \cdot l_3}\right)$$

$$\theta_2 = \frac{\pi}{2} - \arccos\left(\frac{\left|\overrightarrow{O_2O_6}\right|^2 + {l_2}^2 - {l_3}^2}{2 \cdot l_2 \cdot \left|\overrightarrow{O_2O_6}\right|}\right) - \vartheta$$

$$\vartheta = \operatorname{asin}\left(\frac{x_{O_6} - l_1}{|\overrightarrow{O_2 O_6}|}\right)$$

As $\theta_2 = 0$ because of the assumption made before, now only the angles of the 5th and 6th joints need to be computed to provide the desired orientation of the end effector. This task can be easily mapped to finding Euler angles $\alpha, \beta$ and $\gamma$ that would rotate the coordinate frame $O_5 X_5 Y_5 Z_5$ to the frame $O_7 X_7 Y_7 Z_7$, where $X_7$ and $Y_7$ are arbitrary directed. The solution of this task is well known and we obtain, that

$$\theta_5 = \operatorname{atan2}\left(y_{Z_7^5}, x_{Z_7^5}\right)$$

$$\theta_6 = \operatorname{acos}\left(z_{Z_7^5}\right),$$

where $Z_7^5$ is the vector $\overrightarrow{Z_7}$ observed from the coordinate frame $O_5 X_5 Y_5 Z_5$.

After the desired angle values have been found, they should be validated against admissible joint limits. If the test has been successfully passed, the calculated joint values are returned as output of IKS. In case of a fail the inverse kinematics task is considered as unsolvable.

### 3.2.4. Mapping configuration to workspace

To map a configuration to workspace we used the same method, as Kunz et al. in [**46**]. First a big number of points is distributed uniformly within the body of the robot links. Then it is computed which cell collides with each point in the current configuration of the robot. The links of the manipulator are approximated with cylinders.

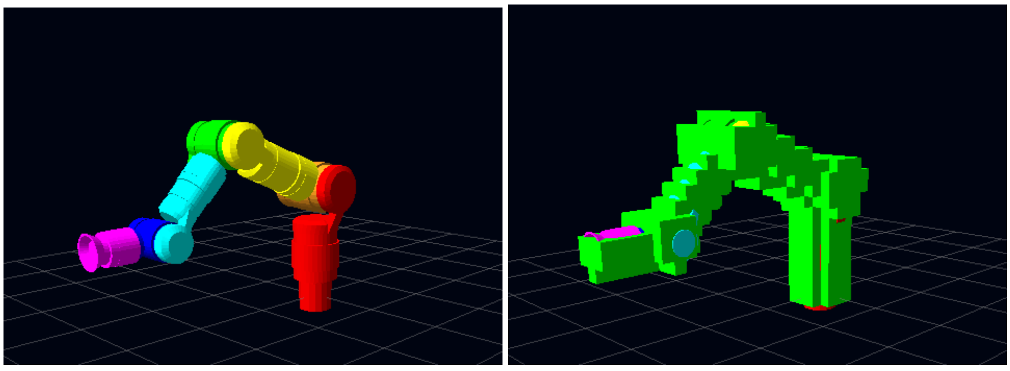


Fig. 26. An example of configuration of a manipulator and its mapping to workspace cells (cell size – 3x3x3 cm; robot's body is sampled with 4525 points).

To provide good coverage of robot's body, the sampling points are located within vertices of a grid. Kunz et al. demonstrate, that there is no much improvement if the grid resolution is less than $\frac{c}{\sqrt{2}}$, where $c$ is the size of workspace cells. In this project the same estimate was used and it appears to be enough to map the whole manipulator to the workspace cells (Fig. 26).

## 3.3. Graph search and obstacle handling

In this chapter the functionality of the graph search algorithm is presented. The graph search runs online and guides the functioning of the task-level controller.

### 3.3.1. Obstacle handling

The first thing that is done after the planner starts is checking for obstacles. In this work we do not consider how the information about obstacle location is obtained. In general, obstacles can be detected with a set of videocameras or a laser range finder. The only data that is required for our framework is the cells of the workspace occupied by obstacles. Based on this information the configurations stored in the roadmap that collide with these cells are invalidated and thus are excluded from the search.

If an obstacle moves, only those cells that change their state with respect to the previous position are checked. If a newly invalidated cell collides with a configuration contained in a node that is present in the currently executed path, the whole path is invalidated and the graph search restarts.

### 3.3.2. Graph search

After currently blocked parts of the roadmap have been invalidated and the start and goal positions of the end effector have been connected to the roadmap, we can do the actual search for a path. A well-known A* search algorithm [**49**] is used to search the graph. It is a heuristic graph search algorithm: an A* search is "guided" by a heuristic function. A heuristic function *h(v)* is the one which estimates the cost from a non-goal state *(v)* in the graph to some goal state. Intuitively, A* follows paths (through the graph) to the goal that are estimated by the heuristic function to be the best. As heuristic function in this project the distance from the current node to the goal end effector position was used. Thus, since all edges have the same weights, the A* search finds the shortest path from the initial node to the goal one.

Each node considered by the search algorithm is examined to be valid and task-consistent. It is done according to the algorithm presented in Fig. 27.

The position constraints are checked first. If the node's position satisfies them, `taskConsistent` is initialized with *true* value. Then the desired orientation vector `D` is computed based on the task for the end effector. It is normalized so that |`D`|=1. Then we define in which directions along the axes of the inertial coordinate frame this vector mostly extends and check if the configurations corresponding to these directions within the current node exist (i.e. a valid solution was found by the inverse kinematics solver in the offline stage) and do not collide with the obstacles. If this is true, then it is assumed that the robot can at least approximately fulfil the task constraints when coming to this node.

```
bool examineNode (node n)
{
   bool taskConsistent = CheckPosition(n);
   If (OrientationTaskSet == true)
   {
      Vector3 D = CalculateDesiredOrientation(n);
      if (D.x > t)
         taskConsistent = taskConsistent and n.configuration(1).exist and n.configuration(1).valid;
      if (D.x < -t)
         taskConsistent = taskConsistent and n.configuration(2).exist and n.configuration(2).valid;
      if (D.y > t)
         taskConsistent = taskConsistent and n.configuration(3).exist and n.configuration(3).valid;
      if (D.y < -t)
         taskConsistent = taskConsistent and n.configuration(4).exist and n.configuration(4).valid;
      if (D.z > t)
         taskConsistent = taskConsistent and n.configuration(5).exist and n.configuration(5).valid;
      if (D.z < -t)
         taskConsistent = taskConsistent and n.configuration(6).exist and n.configuration(6).valid;
      return taskConsistent;
   }
}
```

Fig. 27. Algorithm to define task-consistency of a node.

For a normalized vector D the minimum possible value of the prevailing component in the vector is found when the vector has all three components equal. Then $\min\left(\max_{i=1,2,3}|D_i|\right) = \frac{1}{\sqrt{3}} \approx 0.577$. In all other cases, at least one of the components would be more than this value. Thus, the threshold parameter t from Fig. 27 should be no greater than this value (otherwise, some desired orientation vectors would not be tested and a task-inconsistent node could be included in the path). In the current project t=0.55.

To sum up, with the proposed algorithm the planner searches for a collision-free path of the end effector in the workspace that can be followed in approximately task-consistent manner.

### 3.3.3. Goal point updating and failure detection

After a valid, approximately task-consistent path has been found, it needs to be followed by manipulator. It is done with a common waypoint guidance procedure. The position of first node in the path is fed to the task-level controller (assuming, that it already tries to fulfil orientation constraints). The robot starts moving and when the end effector comes close to the first node, the goal position is updated and the robot moves to the next node. The circle of acceptance is set to $c = 0.5g$, where $g$ is the grid resolution. This process is repeated until the end effector comes to the last point in the path.

There might be situations when the task level controller fails to execute the path. In this case failure detection mechanism, similar to one implemented in *elastic roadmaps*, activates. If the system detects that no progress has been made towards the goal, the current goal node is considered as invalid and the planner tries to find a new path around it. This is the failure of the first type. If no new path found, it means that the planner cannot solve prescribed task. This can refer either to framework's incompleteness (in this case a valid solution can be found with a different approach) or to the fact, that the task is impossible for the robot. To increase the probability to find a valid path again, the nodes which were invalidated in previous runs are revalidated after several unsuccessful replanning attempts. This allows to react to changes in the environment and to find a path that was blocked by obstacles before and becomes free after the obstacle moves.

In *elastic roadmaps* failures of one more type are considered when the task constraints are violated during path following. In the current implementation of our framework this situation is not identified as a failure. This feature can be easily integrated later and its necessity should be defined by system designer.

## 3.4. Task-level controller

Task-level controller is the key component of the framework. It is a convenient and powerful tool for generating multi-objective behaviour for robotic systems using nullspace projections. Instead of specifying explicit joint trajectories, task-level control permits control of the manipulator in operational space, greatly facilitating programming and task specification for kinematically redundant robots.

This part of the thesis and the implementation of the task-level controller are mostly taken from the last semester project [**4**]. However, the set of behaviours and their hierarchy (chapter 3.5) were changed and redesigned to satisfy the needs of the current framework.

In this section, we, first, present principles of the task-level controller functioning and then give algorithms to compute Jacobian and joint-space inertia matrix required for controller operation.

### 3.4.1. Task-level controller

Dynamics of a robot in configuration space is described by the following equation:

$$H(q)\ddot{q} + C(q,\dot{q}) + g(q) = \Gamma + F_{ext},$$

where $\Gamma$ is a vector of joint generalized forces whose components represent torques for revolute joints and forces for translation joints, $q$ – a vector of joint coordinates, $H(q)$ – joint-space inertia matrix (analogue of mass in the 2nd Newton law), $C(q,\dot{q})$ – a vector representing Coriolis and centrifugal forces appearing due to simultaneous movement of several joints, $g(q)$ – gravity force vector, $F_{ext}$ – a vector of external forces including friction. For simplification it will be assumed from now on that there are no external forces acting on the system. Therefore, if we have a vector of desired joint accelerations $\ddot{q}_{des}$, by applying a generalized force vector

$$\Gamma = H(q)\ddot{q}_{des} + C(q,\dot{q}) + g(q)$$

we can provide dynamically-consistent control of the robot and achieve that $\ddot{q} = \ddot{q}_{des}$. This is called feedback linearization as all the system nonlinearities represented with the matrices $H, C$ and $g$ are compensated with the feedback law.

It is possible to map this control law from configuration to operational space. For this we need to choose an operational point in operational space $x(q)$. It can be an end effector of a manipulator or any point of the robot whose coordinates we want to control. After choosing it we can derive a Jacobian of this point

$$J_t = \frac{\partial x(q)}{\partial q}.$$

Jacobian is a matrix representing how change of joint coordinates affects the coordinates of the operational point in operational space. So, if the robot has a task that is formulated in operational space with respect to an operational point, a desired acceleration vector $\ddot{x}_{des}$ corresponding to this task can be derived. The desired acceleration can be achieved if the generalized force vector is computed in accordance with the following equations:

$$\Gamma_{task} = J_t^T F_t,$$

$$F_t = \Lambda_t \ddot{x}_{des} + \mu_t + \rho_t,$$

where $F_t$ is operational-space force acting at the operating point to provide $\ddot{x}_{des}$; $\Lambda_t, \mu_t$ and $\rho_t$ are the inertia matrix, the vector of velocity-product term and the vector of gravitational forces respectively mapped to operational space. The mapping is performed by the following formulae:

$$\Lambda_t = (J_t H^{-1} J_t^T)^{-1},$$

$$\mu_t = \Lambda_t\left(J_t H^{-1} C\dot{q} - \dot{J}_t\dot{q}\right),$$

$$\rho_t = \Lambda_t J_t H^{-1} g.$$

It is necessary to mention that all these parameters differ for different operational points.

If a robot has several tasks that should be performed simultaneously, a generalized force vector $\Gamma_{task\ i}$ is computed for every task and these vectors must be combined into one resulting vector. Simple summing of the vectors does not provide good results as tasks may contradict each other and result in a faulty movement. A method of cascading multiple tasks using nullspaces presented in [**3**] is used in the suggested framework and implements the core functionality of the task-level controller. The work in [**3**] extends approach from [**2**], where only two tasks were combined.

A nullspace of a Jacobian is the space orthogonal to the one spanned by Jacobian and will be of rank $N_{DOF} - rank(J_t)$, where $N_{DOF}$ is the number of degrees if freedom of a robot. The nullspace is represented with square matrix $N$ of size $N_{DOF} \times N_{DOF}$ which is used to project generalized forces from tasks with lower priorities to the nullspace of the task with highest priority. In other words, in case of two tasks (a primary and a secondary) the generalized force vector $\Gamma_\Sigma = \Gamma_{prim} + N_{prim}^T \Gamma_{sec}$ will be consistent with the primary task for any arbitrary generalized force vector from the secondary task. Therefore, the robot would try to accomplish the secondary task while preserving full execution of the primary task. The term $N_{prim}^T \Gamma_{sec}$ is the projection of secondary-task generalized force vector to the nullspace of the primary task.

The presented technique can be expanded to cascade multiple tasks:

$$\Gamma_\Sigma = \Gamma_1 + N_1^T\left(\Gamma_2 + N_2^T\left(\Gamma_3 + N_3^T(\Gamma_4 + \ \dots\ N_{n-1}^T\Gamma_n)\right)\right) = \Gamma_1 + \Gamma_{2|prec(2)} + \Gamma_{3|prec(3)} + \dots + \Gamma_{n|prec(n)}.$$

This equation ensures that the generalized force vector from the tasks with lower priority will be consistent with all tasks with higher priority. The index $prec(i)$ means that the term is consistent with all tasks preceding the task $i$. The task with highest priority is not projected to any nullspace and is executed without any changes.

A nullspace of an individual task is calculated with the following equations:

$$N_t = I - \bar{J}_t J_t = I - (H^{-1} J_t \Lambda_t) J_t.$$

Here $I$ is identity matrix of the same size as $N_t$; $\overline{J_t}$ is dynamically-consistent generalized inverse of $J_t$; the current representation $\overline{J_t} = H^{-1}J_t\Lambda_t$ corresponds to the solution that minimizes the robot's instantaneous kinetic energy.

Projection of a generalized force vector into a nullspace not only makes the vector task-consistent, but also scales it considerably, which can significantly impair task performance. To avoid this the generalized force vector of a task should be defined with the following algorithm:

$$J_{k|prec(k)} = J_k N_{prec(k)}$$

$$F_{k|prec(k)} = \Lambda_{k|prec(k)}\ddot{x}_{des} + \mu_{k|prec(k)} + \rho_{k|prec(k)}$$

$$\Gamma_{k|prec(k)} = J_{k|prec(k)}^T F_{k|prec(k)}$$

The terms $\Lambda_{k|prec(k)}$, $\mu_{k|prec(k)}$ and $\rho_{k|prec(k)}$ are calculated with the formulas given above substituting $J_t$ with $J_{k|prec(k)}$.

A problem may arise while calculating $\Lambda_{k|prec(k)} = \left(J_{k|prec(k)}H^{-1}J_{k|prec(k)}^T\right)^{-1}$. The nullspace matrix $N_{prec(k)}$ in many cases can be not full-rank. Therefore, the task-consistent Jacobian $J_{k|prec(k)}$ can also lose rank and the matrix inversion computing $\Lambda_{k|prec(k)}$ may become impossible. This basically means that the task is not feasible within the current architecture. However, usually the secondary tasks do not have to be fulfilled completely as they have some supporting function, so the part of the task that is feasible should still be used. The feasible movement can be found with singular-value decomposition (SVD) of $\Lambda_{k|prec(k)}^{-1}$ and analysis of its singular values. With SVD the matrix can be factorized into the following form:

$$\Lambda_{k|prec(k)}^{-1} = J_{k|prec(k)}H^{-1}J_{k|prec(k)}^T = \begin{pmatrix} U_{r(k)} & U_{n(k)} \end{pmatrix} \begin{pmatrix} \Sigma_{r(k)} & \\ & 0 \end{pmatrix} \begin{pmatrix} U_{r(k)}^T \\ U_{n(k)}^T \end{pmatrix},$$

where $\Sigma_{r(k)}$ is a diagonal matrix of non-zero eigenvalues, and $U_{r(k)}$ and $U_{n(k)}$ are matrices corresponding to non-zero and zero eigenvectors, respectively. As some eigenvalues are zero, it is not possible to fully control $\ddot{x}_{des}$. However, by choosing the control input

$$F_{k|prec(k)} = \left(U_{r(k)}\Sigma_{r(k)}U_{r(k)}^T\right)^{-1}\ddot{x}_{des} + \mu_{k|prec(k)} + \rho_{k|prec(k)}$$

we accomplish dynamic decoupling in the controllable directions according to the projection $U_{r(k)}(\ddot{x} = \ddot{x}_{des})$, where $U_{r(k)}$ defines these directions.

In [**3**] it is also shown that a nullspace resulting from several cascaded tasks can be calculated as follows:

$$N_k = N_1 N_2 \cdot \ldots \cdot N_{k-1} = I - \sum_{i=1}^{k-1} \bar{J}_{i|prec(i)} J_{i|prec(i)}$$

Therefore, $N_k = N_{k-1} - \bar{J}_{k|prec(k)} J_{k|prec(k)}$.

To summarize all these calculations, here is the overall algorithm how the tasks should be iteratively combined one after another:

1) Initialize the task-level controller; initial nullspace $N_0 = I$;
2) Choose the task;
3) Determine the task Jacobian $J_t$ and the desired acceleration vector $\ddot{x}_{des}$;
4) Get the nullspace resulting from all tasks with higher priority $N_{prec(k)}$;
5) Calculate $J_{k|prec(k)}$ and $F_{k|prec(k)}$ ($\Lambda_{k|prec(k)}$ should be computed using SVD-method);
6) Calculate $\Gamma_{k|prec(k)}$;
7) Calculate the nullspace that combines the nullspace from all the higher-priority tasks and the nullspace from the current task ($N_k = N_{k-1} - \bar{J}_{k|prec(k)} J_{k|prec(k)}$);
8) Repeat steps (2)-(7) for all tasks;
9) Sum up the resulting generalized force vectors $\Gamma_\Sigma = \sum_{k=1}^{n} \Gamma_{k|prec(k)}$ .

### 3.4.2. Jacobian computation

Task-level controller uses Jacobian matrices to calculate generalized torque vectors and nullspaces of tasks. The algorithm for Jacobian computation is presented in this section.

Jacobian of a robot is a matrix representing interdependency between velocities in operational and configuration spaces. Linear and angular velocities of an operational point can be expressed through joint velocities with the following equation:

$$\begin{bmatrix} v_O \\ \omega_O \end{bmatrix} = J(q)\dot{q}$$

In general case

$$J = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \cdots & \frac{\partial x}{\partial q_n} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \cdots & \frac{\partial y}{\partial q_n} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \cdots & \frac{\partial z}{\partial q_n} \\ \frac{\partial \omega_x}{\partial q_1} & \frac{\partial \omega_x}{\partial q_2} & \cdots & \frac{\partial \omega_x}{\partial q_n} \\ \frac{\partial \omega_y}{\partial q_1} & \frac{\partial \omega_y}{\partial q_2} & \cdots & \frac{\partial \omega_y}{\partial q_n} \\ \frac{\partial \omega_z}{\partial q_1} & \frac{\partial \omega_z}{\partial q_2} & \cdots & \frac{\partial \omega_z}{\partial q_n} \end{bmatrix}$$

Thus for one operational point maximal size of $J$ is *6xn*, where *n* is number of joints. The top 3 rows correspond to linear velocities, the bottom 3 rows – to angular velocities. Each column of Jacobian corresponds to one joint and shows how the coordinates of the operational point change if this joint moves. Therefore it is convenient to compute Jacobian column by column considering the joints separately. Each column then can be represented as $J_c^i = \begin{bmatrix} J_v^i \\ J_\omega^i \end{bmatrix}$.

Exact formulae for computation of $J_c^i$ depend on type of joints. The robot at hand has only revolute joints, and they will be discussed in details. The formulas given in the remaining of this section and their mathematical proof is given in [**50**].

In Denavit-Hartenberg notation revolute joint $i$ can rotate around z-axis of $i-1$ coordinate frame. In this case the orientation of the operational point is affected directly and $J_\omega^i = \boldsymbol{z}_{i-1}$. Change of linear coordinates depends on the position of the operational point relative to the rotation axis: the further it is from the axis, the bigger linear velocity the operational point can reach. Thus

$$J_v^i = \boldsymbol{z}_{i-1} \times (\boldsymbol{o}_{op} - \boldsymbol{o}_{i-1})$$

where $\boldsymbol{o}_{op}$ and $\boldsymbol{o}_{i-1}$ are the positions of the operational point and the origin of $i-1$ coordinate frame viewed from the world coordinate frame (Fig. 28.b).

If the operational point is such, that it cannot be affected by a joint (for instance, it is closer to the root of manipulator's kinematic chain than the joint), the corresponding column of Jacobian is set to zero.

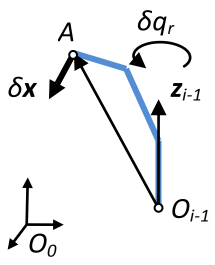


Fig. 28. Instantaneous linear velocity of operational point caused by the change of joint coordinates for a revolute joint ($\delta x$ is perpendicular to the plane formed by $\boldsymbol{z}_{i-1}$ and $\overrightarrow{O_{i-1}A}$ ).

### 3.4.3. Joint-space inertia matrix computation

JSIM is a symmetric positive-definite matrix and it represents inertia properties of the robot. For instance, full kinetic energy of a robot can be specified as

$$K = \frac{1}{2}\dot{q}^T H \dot{q}.$$

The general formula to compute JSIM is

$$H(q) = \sum_{i=1}^{n} \left( m_i {J_{v}}_i^T {J_{v}}_i + {J_{\omega}}_i^T R_i I_i R_i^T {J_{\omega}}_i \right)$$

Here $m_i$ and $I_i$ are the mass and the inertia tensor of *i*-th link of the robot; ${J_{v}}_i$ and ${J_{\omega}}_i$ are the submatrices of the Jacobian corresponding to linear and angular velocities of the operational point respectively; the index *i* here means that the operational point coincides with the origin of the coordinate frame connected to the *i*-th link of the robot; $R_i$ is the transform matrix representing transformation from global coordinate frame to the coordinate frame connected to the *i*-th link of the robot [**50**].

It is obvious that calculation of JSIM is computationally intensive as multiple Jacobians and transform matrices need to be computed. To reduce the number of operations required for dynamics representation of a robot special a framework using *spatial algebra* notation was presented by Roy Featherstone in [**51**]. Part of this framework is the *Composite-Rigid-Body Algorithm* for JSIM calculation which is used in the current implementation. In this section we give a very short introduction into the spatial algebra first and then present the algorithm itself.

#### *Spatial algebra quantities*

A body in 3D space has 6 degrees of freedom, but traditionally linear and angular parameters are analyzed separately. Spatial algebra uses 6-dimentional vectors and matrices to represent different properties of rigid bodies. Some of them are summarized in Table 1.

| Symbol | Definition | Full representation | Compact representation |
|---|---|---|---|
| $\boldsymbol{v}$ | Velocity of a rigid body | $\boldsymbol{v} = \begin{pmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_O \end{pmatrix}$ | $(\boldsymbol{\omega}; \boldsymbol{v}_O)$ |
| $\boldsymbol{a}$ | Acceleration of a rigid body ($\boldsymbol{a} = \dot{\boldsymbol{v}}$) | $\boldsymbol{a} = \begin{pmatrix} \dot{\boldsymbol{\omega}} \\ \dot{\boldsymbol{v}_O} \end{pmatrix} = \boldsymbol{a}' - \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{\omega} \times \boldsymbol{v}_O \end{pmatrix}$ | $(\dot{\boldsymbol{\omega}}; \dot{\boldsymbol{v}_O})$ |
| $\boldsymbol{I}_O$ | Spatial inertia of a rigid body around point O. | $\boldsymbol{I}_O = \begin{pmatrix} \bar{\boldsymbol{I}}_O & m\boldsymbol{S}(\boldsymbol{c}) \\ m\boldsymbol{S}(\boldsymbol{c})^T & m\boldsymbol{1} \end{pmatrix}$ | $(\bar{\boldsymbol{I}}_O; \boldsymbol{h}; m)$ |
| ${}^B\boldsymbol{X}_A$ | Coordinate transform from frame A to frame B. | ${}^B\boldsymbol{X}_A = \begin{pmatrix} {}^B\boldsymbol{R}_A & \boldsymbol{0} \\ {}^B\boldsymbol{R}_A\boldsymbol{S}({}^B\boldsymbol{p}_A)^T & {}^B\boldsymbol{R}_A \end{pmatrix}$ | $({}^B\boldsymbol{R}_A; {}^B\boldsymbol{p}_A)$ |

Table 1. Some elements of the spatial algebra and their compact representation.

Velocity of a rigid body can be expressed as a pair of 3D vectors $\boldsymbol{\omega}$ and $\boldsymbol{v}_O$ which specify the body's angular velocity and the linear velocity of a body-fixed point currently coinciding with point O (location of the point O is specified in the fixed coordinate frame). These two vectors are united to form the spatial velocity vector $\boldsymbol{v}$.

Spatial acceleration is defined as the rate of change of spatial velocity. Unfortunately, this means that spatial acceleration differs from the classical textbook definition of rigid-body acceleration, which shall be called as *classical acceleration*. Essentially, the difference can be summarized as follows:

$$\boldsymbol{a} = \begin{pmatrix} \dot{\boldsymbol{\omega}} \\ \dot{\boldsymbol{v}_O} \end{pmatrix}, \qquad \boldsymbol{a}' = \begin{pmatrix} \dot{\boldsymbol{\omega}} \\ \dot{\boldsymbol{v}}'_O \end{pmatrix}$$

where $\boldsymbol{a}$ is the spatial acceleration, $\boldsymbol{a}'$is the classical acceleration, $\dot{\boldsymbol{v}_O}$ is the derivative of $\boldsymbol{v}_O$ taking *O* to be fixed in space, and $\dot{\boldsymbol{v}}'_O$ is the derivative of $\boldsymbol{v}_O$ taking *O* to be fixed in the body. Mathematically the difference is expressed with the equation

$$\boldsymbol{a} = \boldsymbol{a}' - \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{\omega} \times \boldsymbol{v}_O \end{pmatrix}.$$

The practical difference is that spatial accelerations are easier to use. For example, if the bodies $B_1$ and $B_2$ have velocities of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ respectively, and $\boldsymbol{v}_{rel}$ is the relative velocity of $B_2$ with respect to $B_1$, then $\boldsymbol{v}_2 = \boldsymbol{v}_1 + \boldsymbol{v}_{rel}$. The relationship between their spatial accelerations is obtained simply by differentiating the velocity formula: $\frac{d}{dt}(\boldsymbol{v}_2 = \boldsymbol{v}_1 + \boldsymbol{v}_{rel}) \;=>\; \boldsymbol{a}_2 = \boldsymbol{a}_1 + \boldsymbol{a}_{rel}$. Thus, spatial accelerations are composed by addition, exactly like velocities, and there are no Coriolis or centrifugal terms to worry about.

Spatial inertia around point O is a 6x6 square matrix and is computed according Table 1. Here $m$ is the mass of the body; $\boldsymbol{c}$ is a 3D vector from O to the centre of mass C;

$\boldsymbol{h} = m\boldsymbol{c}$ is an auxiliary vector without specific physical meaning; $\bar{\boldsymbol{I}}_O = \bar{\boldsymbol{I}}_{cm} + m\boldsymbol{S}(\boldsymbol{c})\boldsymbol{S}(\boldsymbol{c})^T$ is the rotational inertia tensor of the rigid body around O; $\bar{\boldsymbol{I}}_{cm}$ – rotational inertia tensor of the rigid body around the centre of mass; $\boldsymbol{S}(\boldsymbol{c})$ is a skew-symmetric matrix formed from the 3D vector $\boldsymbol{c}$ according to the following equation:

$$\boldsymbol{S}(\boldsymbol{c}) = \begin{pmatrix} 0 & -c_z & c_y \\ c_z & 0 & -c_x \\ -c_y & c_x & 0 \end{pmatrix}.$$

It can be shown that for any 3D vector $\boldsymbol{p}$ the following is true: $\boldsymbol{S}(\boldsymbol{c})\boldsymbol{p} = \boldsymbol{p} \times \boldsymbol{c}$.

Coordinate transform from frame A to frame B is a 6x6 square matrix and it is computed according to the Table 1. Here ${}^{B}\boldsymbol{R}_A$ is 3x3 rotation matrix transforming coordinates from frame A to frame B; ${}^{B}\boldsymbol{p}_A$ Is a 3D vector defining location of the origin of the frame B relative to the origin of the frame A, expressed in coordinates of A.

### *Spatial algebra formulae*

As can be seen from Table 1, some of spatial quantities have compact representation, so it is not necessary to store 36 elements of 6x6 spatial matrices, but only 12 and 13 elements to represent coordinate transform and spatial inertia respectively. This also simplifies operations with these quantities greatly. The compact representation of spatial arithmetic formulae is given in Table 2.

| Expression | Meaning | Compact value |
|---|---|---|
| $\boldsymbol{Xv}$ | Expressing $\boldsymbol{v}$ in coordinates of another frame | $(\boldsymbol{R\omega}; \boldsymbol{R}(\boldsymbol{v}_O - \boldsymbol{p} \times \boldsymbol{\omega}))$ |
| $\boldsymbol{X}^{-1}$ | Coordinate transform inversion | $(\boldsymbol{R}^T; -\boldsymbol{Rp})$ |
| $\boldsymbol{X}_1\boldsymbol{X}_2$ | Combining two coordinate transforms | $(\boldsymbol{R}_1\boldsymbol{R}_2;\ \boldsymbol{p}_2 + \boldsymbol{R}_2^T\boldsymbol{p}_1)$ |
| $\boldsymbol{I}_{O1} + \boldsymbol{I}_{O2}$ | Summing inertias around the same point O from several bodies | $(\bar{\boldsymbol{I}}_{O1} + \bar{\boldsymbol{I}}_{O2};\ \boldsymbol{h}_1 + \boldsymbol{h}_2; m_1 + m_2)$ |
| $\boldsymbol{X}^T\boldsymbol{IX}$ | Expressing spatial inertia in coordinates of another frame | $(\boldsymbol{R}^T\bar{\boldsymbol{I}}_O\boldsymbol{R} - \boldsymbol{S}(\boldsymbol{p})\boldsymbol{S}(\boldsymbol{R}^T\boldsymbol{h}) - \boldsymbol{S}(\boldsymbol{R}^T\boldsymbol{h} + m\boldsymbol{p})\boldsymbol{S}(\boldsymbol{p});$ $\boldsymbol{R}^T\boldsymbol{h} + m\boldsymbol{p};\ m)$ |

Table 2. Effective representation of spatial algebra formulae.

### *Composite-Rigid-Body Algorithm*

Composite-Rigid-Body Algorithm is used to effectively calculate JSIM. Generally speaking it can be applied to a kinematic tree with multiple branches (for example, a humanoid robot with several limbs). Fig. 29 represents pseudocode for the algorithm when applied to a robot without kinematic branching.

The following notations are used in the algorithm:

- $n$ – number of joints;
- $\boldsymbol{I}_i$ – spatial inertias of separate links given about the origins of coordinate frames attached to the respective links;
- ${}^{i}\boldsymbol{X}_{i-1}$ – coordinate transforms from the frame attached to the link $i$ to a frame attached to the link preceding the link $i$;
- $\boldsymbol{\Phi}_i$ – a matrix representing unconstrained degrees of freedom of joint $i$. For example, for a revolute joint rotating around z-axis $\boldsymbol{\Phi}_{rev} = (0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0)^T$; for a spherical joint allowing rotation around three axes $\boldsymbol{\Phi}_{sph} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}^T$; for a prismatic joint moving along z-axis $\boldsymbol{\Phi}_{pr} = (0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1)^T$; for cylindrical joint (rotation around z-axis and movement along z-axis) $\boldsymbol{\Phi}_{cyl} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T$;
- $\boldsymbol{I}_i^C$ – inertia of a composite rigid body formed by the rigid assembly of the joint $i$ and all the links that are closer to the end of kinematic branch than the joint $i$;
- $\boldsymbol{F}$ – local variable.

So the algorithm iteratively calculates spatial inertias of composite rigid bodies composed of several links (for instance, if $i = n - 1$ the composite body consists of the two last links of the robot and it is assumed that they do not move relative to each other) and expresses the inertia in coordinates of corresponding frame. Then with the help of matrices $\boldsymbol{\Phi}_i$ the exact terms of spatial inertia matrix corresponding to unconstrained degrees of freedom of respective joints are chosen to be components of JSIM.

Although this algorithm requires more design efforts (developing background for implementation of spatial algebra), it provides better performance as all the calculations can be performed in compact and effective form.

```
$inputs: \boldsymbol{I}_i, n, {}^{i}\boldsymbol{X}_{i-1}, \boldsymbol{\Phi}_i$
$\boldsymbol{H} = \boldsymbol{0};$
for $i = 1$ to $n$ do
        $\boldsymbol{I}_i^C = \boldsymbol{I}_i;$
end for;
for $n$ to $i = 1$ do
        $\boldsymbol{F} = \boldsymbol{I}_i^C \boldsymbol{\Phi}_i;$
        $H_{ii} = \boldsymbol{\Phi}_i^T F;$
        if $i \neq 1$ then
          $\boldsymbol{I}_{i-1}^C = \boldsymbol{I}_{i-1}^C + {}^{i}\boldsymbol{X}_{i-1}^T \boldsymbol{I}_i^C \, {}^{i}\boldsymbol{X}_{i-1};$
        $j = i;$
        while $j \neq 1$ do
          $\boldsymbol{F} = {}^{j}\boldsymbol{X}_{j-1} \boldsymbol{F};$
                $j$--;
                $\boldsymbol{H}_{ij} = \boldsymbol{F}^T \boldsymbol{\Phi}_j;$
                $\boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^T;$
        end while;
end for;
```

Fig. 29. Composite-Rigid-Body Algorithm for calculating JSIM of a non-branching robot.

## 3.5. Task hierarchy and design of individual tasks

### 3.5.1. Task prioritization in the task-level controller

The task-level controller in our implementation has the following task hierarchy (notation is taken from [**44**]):

$$\varphi_{stop} \lhd \varphi_{posture} \lhd \varphi_{global} \lhd \varphi_{task} \lhd \varphi_{collision} \lhd \varphi_{kinematic}.$$

Here the notation $\varphi_1 \lhd \varphi_2$ means that $\varphi_2$ has higher priority than $\varphi_1$. $\varphi_{kinematic}$ is a task that prevents reaching joint limits by the robot; it has the highest priority in all task-level controllers and is executed under any circumstances. $\varphi_{collision}$ provides obstacle avoidance; $\varphi_{task}$ represents the main manipulation task; $\varphi_{global}$ is responsible for global robot motion towards the next end effector waypoint; $\varphi_{posture}$ is a secondary task specifying desired configuration of the robot; $\varphi_{stop}$ is the lowest-priority task and constrains all degrees of freedom unaffected by all other tasks.

The two tasks with the highest priority provide that the manipulator operates safely. Prioritization of $\varphi_{task}$ and $\varphi_{global}$ depend on the decision of the system designer or on the manipulation task of the robot. $\varphi_{posture}$ provides correspondence between offline roadmap generation and online path following.

### 3.5.1. Some notes on the implementation of tasks

Some tasks may require that not all joints should be affected. For example, $\varphi_{kinematic}$ is not always active but only when a joint is close to its limit and only this joint should be affected while the others may move arbitrary. In this case we may assume that the task works in configuration space and, therefore, no $\ddot{x}_{des}$ for operation point can be computed. Instead the desired acceleration is computed in configuration space ($\ddot{q}_{des}$ instead of $\ddot{x}_{des}$) and directly transformed to a generalized force vector as $\Gamma = H(q)\ddot{q}_{\,des} + C(q,\dot{q}) + g(q)$. The task Jacobian, required for computing the nullspace of the task, is formed from the end effector Jacobian. Only the columns corresponding to the joints that are directly controlled by the task are used; columns corresponding to other joints are set to zero. Based on the task Jacobian the matrices $\Lambda_{k|prec(k)}$, $\bar{J}_{k|prec(k)}$ and the resulting nullspace are computed in the same way as for operational-space tasks.

A manipulation task working in operation space usually specifies not all coordinates, but only some of them. For instance, obstacle avoidance does not restrict operation point orientation. A task Jacobian at most can have 6 rows (3 for linear and 3 for angular coordinates). To match a task Jacobian to its task, only the rows of Jacobian corresponding to operational coordinates that are of interest for current task should be used in computations while others may be disregarded.

There can be periods of time when a task is inactive and it does not need to move its operation point (or joints in configuration space) or produce any activity. In this case its resulting generalized force vector $\Gamma_{k|prec(k)}$ is set to zero and the overall nullspace remains unchanged ($N_k = N_{k-1}$). Activation or deactivation of a task may be done either inside or outside of the task. For example, $\varphi_{kinematic}$ is always turned on, but it is active only when at least one of the robot joints is close to its limit. In this case it is activated internally after checking current values of the joint angles. Similarly, $\varphi_{collision}$ needs to check if the robot collides with obstacles first and make a decision about activating itself. On other hand, $\varphi_{task}$ does not check anything internally and can be activated or deactivated only from outside.

The core feature of each task is determination of $\ddot{q}_{des}$ or $\ddot{x}_{des}$. In the current project it is computed with simple PD-controllers according to the following equations:

$$\dot{x}_{des} = \frac{k_p}{k_v}(x_{des} - x)$$

$$v = \min\left(1, \frac{V_{max}}{\|\dot{x}_{des}\|}\right)$$

$$\ddot{x}_{des} = -k_v(\dot{x} - v\dot{x}_{des})$$

Here $x$ and $\dot{x}$ are actual coordinates and velocity of the operational point; $x_{des}$, $\dot{x}_{des}$ – desired coordinates and velocity of the operational point; $k_p$ and $k_v$ – gain coefficients. The velocity vector $\dot{x}$ is in fact controlled to be pointed toward the goal position $x_{des}$ while its magnitude is limited to $V_{max}$. It is equivalent to building a potential field with an attraction point at $x_{des}$ and going to it in the shortest way. For tasks acting in configuration space $\ddot{q}_{des}$ is computed in a similar way substituting operational-space coordinate vectors with joint-space ones.

### 3.5.2. Design of tasks

Each task has the following specific features: task's goal; which space, operational or configuration, it operates in; operation point; when the task is active; affected joints; algorithm to define $x_{des}$ (or $q_{des}$). In this subsection design decision concerning each task individually will be presented.

The tasks $\varphi_{global}$ and $\varphi_{kinematic}$ are similar to ones implemented last semester. The tasks $\varphi_{task}$, $\varphi_{collision}$, $\varphi_{posture}$ and $\varphi_{stop}$ were designed during the current project.

### *Avoiding joint limits ($\varphi_{kinematic}$)*

**Operational or configuration space:** Configuration space

**Affected joints:** All joints that are currently close to their limits

**Activated:** When a joint is within certain range $q_c$ from its maximal or minimal limit.

**Working algorithm:**

$$\begin{cases} q_{des_i} = q_{max_i} - q_{c_i}, \text{if } q_i > q_{max_i} - q_{c_i} \\ q_{des_i} = q_{min_i} + q_{c_i}, \text{if } q_i < q_{min_i} + q_{c_i} \\ \text{inactive for the joint } i, \text{else} \end{cases}$$

where $q_c$ is a range where a joint is considered to be close to its maximal or minimal limit, $i$ is the number of a joint. Value of $q_c$ is defined based on maximum velocity and acceleration of joint's actuator, so that a joint is able to stop before it reaches its physical limit. Such realisation of $\varphi_{kinematic}$ decreases the admissible range of angle values but allows preventing mechanical damage of robot links and actuators.

### *Obstacle avoidance ($\varphi_{collision}$)*

**Operational or configuration space:** Operational space

**Operation point:** End points of manipulator segments

**Affected joints:** All joints preceding the operation point

**Activated:** When robot is within certain distance from an obstacle.

**Working algorithm:**

In the current project for collision avoidance the robot is divided in four segments: segment 0 – links 1 and 2, segment 1 – links 3 and 4, segment 2 – links 5 and 6, segment 3 – link 7 (Fig. 30). The segment 0 is stationary and no collision avoidance measures can be taken. The overall obstacle avoidance task in fact can be divided in several subtasks each responsible for its segment, i.e. each subtask tries to move the corresponding segment away from obstacles. The subtasks are prioritized as $\varphi_{segment\ 3} \lhd \varphi_{segment\ 2} \lhd \varphi_{segment\ 1}$, so that segment 1 has the highest priority. At first it may seem strange to have different priorities for different parts of the robot. This does not mean, however, that certain parts are more likely to collide. This is because higher-priority subtasks affect only part of robot joints, meaning that the nullspace for lower-priority subtasks will remain unaffected for all other joints. For instance, if both segments 1 and 3 are close to obstacles, $\varphi_{segment\ 1}$ will define movement of joints 1 and 2, while $\varphi_{segment\ 3}$ may use all other degrees of freedom (and joints 1 and 2 if they move in the appropriate direction) to prevent a collision of segment 3.

It is very costly to analyze precisely collisions of rigid bodies online. An appropriate approximation of robot's links is needed. In the current framework each segment is represented as a set of spheres located on the main axes of the links of the robot (Fig. 30). An obstacle object within the framework stores the cells that it occupies. Thus, to provide obstacle avoidance we need to ensure that centres of the spheres are far away enough from the cells occupied by obstacles. It is done with the algorithm in Fig. 31.

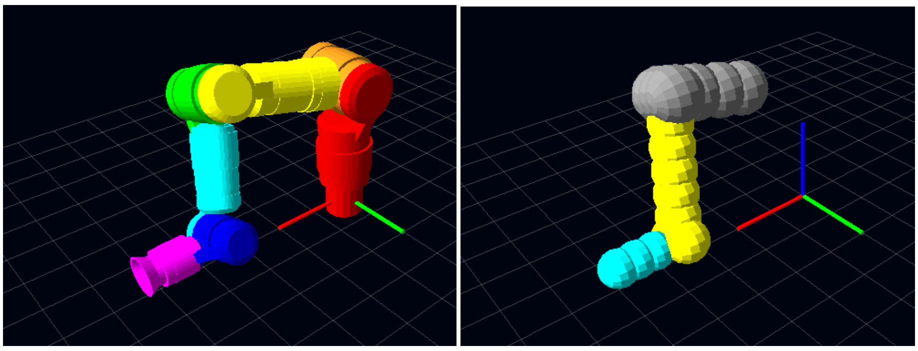


Fig. 30. Approximation of the robot's links with spheres for obstacle avoidance. Left – robot model, right – spheres used in online collision avoidance (grey – segment 1, yellow – segment 2, blue – segment 3).

```
for each obstacle $o_i$
  for each cell $c_j \in o_i$
    if (**CheckCellClose** ($c_j$))
      for each sphere $s_k$
        if **Distance** (**Center** ($s_k$) - **Center** ($c_j$)) < *AllowedRange*
          $force_k$ += **CalculateForce** ($s_k$, $c_j$)
      end
  end
end

for each segment $r_m$
  for each sphere $s_k \in r_m$
    $segmentForce_m$ += $force_k$
  end
end
**CalculateReferenceAcceleration** ()
```

Fig. 31. Algorithm for online collision avoidance

Some comments on the algorithm:

- The operating point for the segment 1 is the origin of joint 4, for the segment 2 – the origin of joint 6, for the segment 3 – the end effector (points $O_4$, $O_6$ and $A$ in Fig. 25).
- Function **CheckCellClose** ($c_j$)) returns *true* if the cell $c_j$ is within 0,3 m from operating points of the 1st and the 2nd segments or within 0,2 m from operating point of the 3rd segment. This quick check allows to filter out not obstacle-free cells that currently far from the robot and cannot obstruct robot's movement.
- The variable *AllowedRange* should consider the size of cells and the safety distance defined by the designer.
- The function **CalculateForce**($s_k$, $c_j$) defines a virtual force that drives the sphere $s_k$ from the cell $c_j$. The force is always directed from the center of $c_j$ to the center of $s_k$ and its absolute value is calculated as

$$|f| = \begin{cases} \frac{9}{R}, & \text{if } l > d \\ \frac{1}{d-l} - \frac{1}{R}, & \text{if } l \leq d < l + R \\ 0, & \text{if } d \geq l + R \end{cases},$$

  where $d$ is the distance between the centres of $s_k$ and $c_j$, $R$ is the desired safety distance, $l = \text{size}(c_j) \cdot \sqrt{3} + 0{,}1R$ – constant considering size of the cells. The graphical representation of $|f|$ is depicted in Fig. 32.
- When summing up the forces from spheres within one segment we do not consider the location of spheres within the segment. Thus, we constrain only position of the operating point and not the orientation of the link it is attached to. Such solution allows to avoid imposing too many constraints on the operating point and to leave more freedom for lower-priority tasks.
- The resulting force actually represents where the operating point should move to get away from obstacles. Thus, we may say, that $segmentForce_m = \dot{x}_{des_m} \cdot k$, where $k$ is some scaling coefficient. If we define $k = \min\left(1, \frac{V_{max_m}}{\|segmentForce_m\|}\right)$, the corresponding desired acceleration $\ddot{x}_{des}$, torque vector $\Gamma$ and resulting nullspace $N$ may be computed with the procedure described in section 3.5.1.

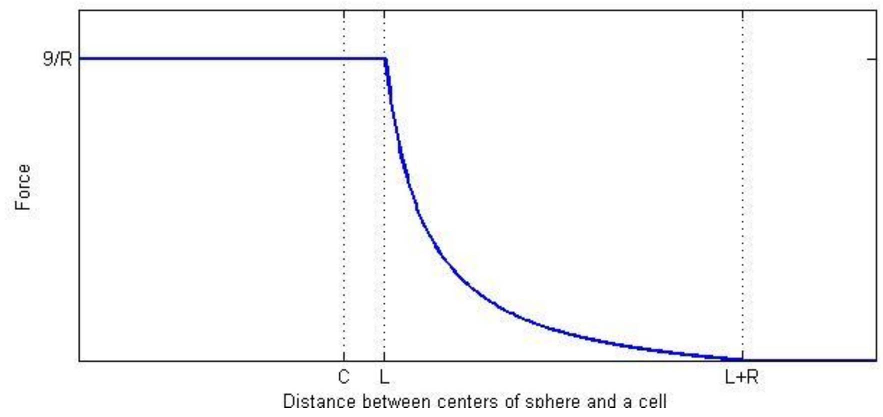


Fig. 32. Dependence of the absolute value of force repulsing a sphere from a cell on the distance between them (C=size$(c_j)$ ).

### *End effector task ($\varphi_{task}$)*

**Operational or configuration space:** Configuration space

**Affected joints:** Joints 5 and 6

**Activated:** When a task for the end effector is specified.

**Working algorithm:**

In this task we consider only orientation of the end effector. Positional constraints for the end effector are fulfilled during graph search stage. Two manipulation tasks involving orientation task constraints are modelled in the current project: preserving constant orientation of the end effector (e.g. while carrying a glass of water) and aiming with the end effector at the same point in the space throughout movement.

Although the tasks are specified in the operational space, it was easier to implement the end effector task in the configuration space. We assume that orientation constraints may be satisfied only by the movement of the joints close to the end effector. Thus, we find the desired angle values for joints 5 and 6 with the same algorithm as implemented in inverse kinematics solver:

$$q_{des_5} = \text{atan2}\,(y_{Z_7^5}, x_{Z_7^5})$$

$$q_{des_6} = \text{acos}\,(z_{Z_7^5}),$$

where $Z_7^5$ $(x_{Z_7^5}, y_{Z_7^5}, z_{Z_7^5})$ is the vector of desired orientation of the end effector observed from the coordinate frame $O_5X_5Y_5Z_5$. If the task is to aim at a point $P$ we need first to calculate $Z_7$ as

$$Z_7 = \frac{\overrightarrow{OP} - \overrightarrow{OA}}{\left|\overrightarrow{OP} - \overrightarrow{OA}\right|}$$

Where point $O$ is the centre of the inertial frame, $A$ is the current position of the end effector.

### *End effector movement ($\varphi_{global}$)*

**Operational or configuration space:** Operational space

**Operation point:** End effector

**Affected joints:** All

**Activated:** When a goal point is updated by the planner.

**Working algorithm:**

This task moves the end effector towards the point specified by planner. If we assume that the goal point is given by $x_{goal}$, then the algorithm from subsection 3.5.1 can be used directly:

$$\dot{x}_{des} = \frac{k_p}{k_v}(x_{goal} - x)$$

$$v = \min\left(1, \frac{V_{max}}{\|\dot{x}_{des}\|}\right)$$

$$\ddot{x}_{des} = -k_v(\dot{x} - v\dot{x}_{des}).$$

### *Secondary configuration task ($\varphi_{posture}$)*

**Operational or configuration space:** Configuration space

**Affected joints:** Joints 3 and 4

**Activated:** Always.

**Working algorithm:**

This task tries to ensure that the current configuration is subject to the same constraints, as configurations found in preprocessing stage during inverse kinematics solving (see subsection 3.2.2). More specifically, it tries to keep the angle value of the joint 3 at zero and prevents assignment of negative angle values to the joint 4. Therefore we impose a soft constraint on the robot’s configuration during movement to provide correspondence between offline preprocessing and online movement.

Mathematically the algorithm can be written as follows:

$$\left[\begin{array}{l} q_{des_3} = 0 \\ \begin{cases} q_{des_4} = 0, \text{ if } q_4 < 0 \\ \text{inactive for the joint } 3, \text{ if } q_4 > 0 \end{cases} \end{array}\right.$$

Vector of desired acceleration $\ddot{q}_{des}$ is then found with a standard procedure.

#### *Task constraining all available degrees of freedom ($\varphi_{stop}$)*

**Operational or configuration space:** Configuration space

**Affected joints:** All

**Activated:** Always.

**Working algorithm:**

This is the lowest-priority task and it affects all joints all the time so the robot never has unconstrained degrees of freedom (the nullspace of the current task is zero). To achieve this $\varphi_{stop}$ tries to stop all joints that move freely in the nullspace of all higher-priority tasks. Thus, the vector of desired joint velocities $\dot{q}_{des} = 0$, and $\ddot{q}_{des} = -k_v\dot{q}$, where $\dot{q}$ is the vector of actual joint velocities.

### 3.5.3. Summary of tasks

The tasks described previously are summarized in the Table 3.

| Notation | Goal | Type | Activation condition | Operational point | Affected joints | Desired position |
|---|---|---|---|---|---|---|
| $\varphi_{kinematic}$ | Avoid joint limits | C | When a joint is within certain range $q_c$ from its maximal or minimal limit. | – | Only those joints that are close to their respective joint limits | $q_{des_i} = q_{max_i} - q_{c_i}$ if $q_i > q_{max_i} - q_{c_i}$; $q_{des_i} = q_{min_i} + q_{c_i}$ if $q_i < q_{min_i} + q_{c_i}$. |
| $\varphi_{collision}$ | Avoid obstacles | O | When robot is within certain distance from an obstacle. | End of the segment that is close to an obstacle | All joints preceding the operational point. | Determined by the direction of the repulsive force. |
| $\varphi_{task}$ | End effector orientation task. Currently 2 options are available: • Keeping constant orientation • Aiming at a certain point of the workspace | C | Activated when a task is specified. | – | Joints 5 and 6 | $q_{des_5} = \text{atan2}\,(y_{Z_7^5}, x_{Z_7^5})$ $q_{des_6} = \text{acos}\,(z_{Z_7^5})$, |
| $\varphi_{global}$ | Moving the end effector to the desired goal point. | O | Activated by the path planner | End effector | All | Specified by the planner |
| $\varphi_{posture}$ | Posture specification | C | Always active | – | Joints 3 and 4 | $\left[\begin{array}{l} q_{des_3} = 0 \\ \begin{cases} q_{des_4} = 0, \text{if } q_4 < 0 \\ \text{inactive for the joint 3, if } q_4 > 0 \end{cases} \end{array}\right.$ |
| $\varphi_{stop}$ | Constraining all free degrees of freedom. | C | Always active | – | All | $q_{des} = q_{current}$ (then $\dot{q}_{des} = 0$) |

Table 3. Individual characteristics of different tasks. In *Type* column *C* corresponds to tasks working in configuration space, *O* – in operational space. The tasks are listed according to their priorities (higher priority first).

## 3.6. Robot simulation model

As already explained in chapter 3.4.1, a dynamic model of a robot is represented with the following equation:

$$H(q)\ddot{q} + C(q,\dot{q}) + g(q) = \Gamma + F_{ext}$$

The term $F_{ext}$ can relate, for example, to the friction in the actuators and gear units. The generalized force vector $\Gamma$ is generated by the task-level controller and now the task is to compute how the robot configuration changes when $\Gamma$ is applied to the robot's actuators. This task is known as forward dynamics calculation.

From the last equation we can derive that $\ddot{q} = H^{-1}(\Gamma + F_{ext} - C - g)$. The joint-space inertia matrix $H$ is always invertible. At the current stage of the project the terms $F_{ext}, C$ and $g$ are not considered yet and their calculation should be implemented in future work. Therefore the only term left is

$$\ddot{q} = H_p{}^{-1}\Gamma.$$

The state of a robot is described with a pair of vectors of space variables $(X_1; X_2)$, where $X_1$ corresponds to the configuration variables $q$ and $X_2$ corresponds to their velocities $\dot{q}$. Thus, system description in state-space is

$$\begin{cases} \dot{X}_1 = X_2 \\ \dot{X}_2 = \ddot{q} \end{cases}.$$

Considering the previous equations we obtain that

$$\begin{cases} \dot{X}_1 = X_2 \\ \dot{X}_2 = H_p{}^{-1}\Gamma \end{cases}$$

This state-space model represents continuous-time system. However, computer simulation can be performed only in discrete time. To define new state of the system after a time step $h$ the Newton integration method is used:

$$\begin{cases} X_1[k] = X_1[k-1] + X_2[k-1]h \\ X_2[k] = X_2[k-1] + (H^{-1}\Gamma)h \end{cases}$$

Another aspect of robot simulation is physical limitations: the values of torques, accelerations and velocities provided by the actuators cannot exceed some certain values. Therefore, $\Gamma$, $\ddot{q}$ and $X_2$ get bounded after their values are computed in our simulation model.

# 4. Implementation

This chapter reveals some implementation details of the current project. It describes the structure of the proposed framework and the tools used during implementation.

## 4.1. Class diagram

The framework is implemented in C++. Fig. 33 and Fig. 34 show simplified class diagram of the offline and online parts of the framework. Functionality of most of the classes is clear from their names and is presented in details in the Design chapter (Chapter 3). For other classes the necessary comments are presented further.

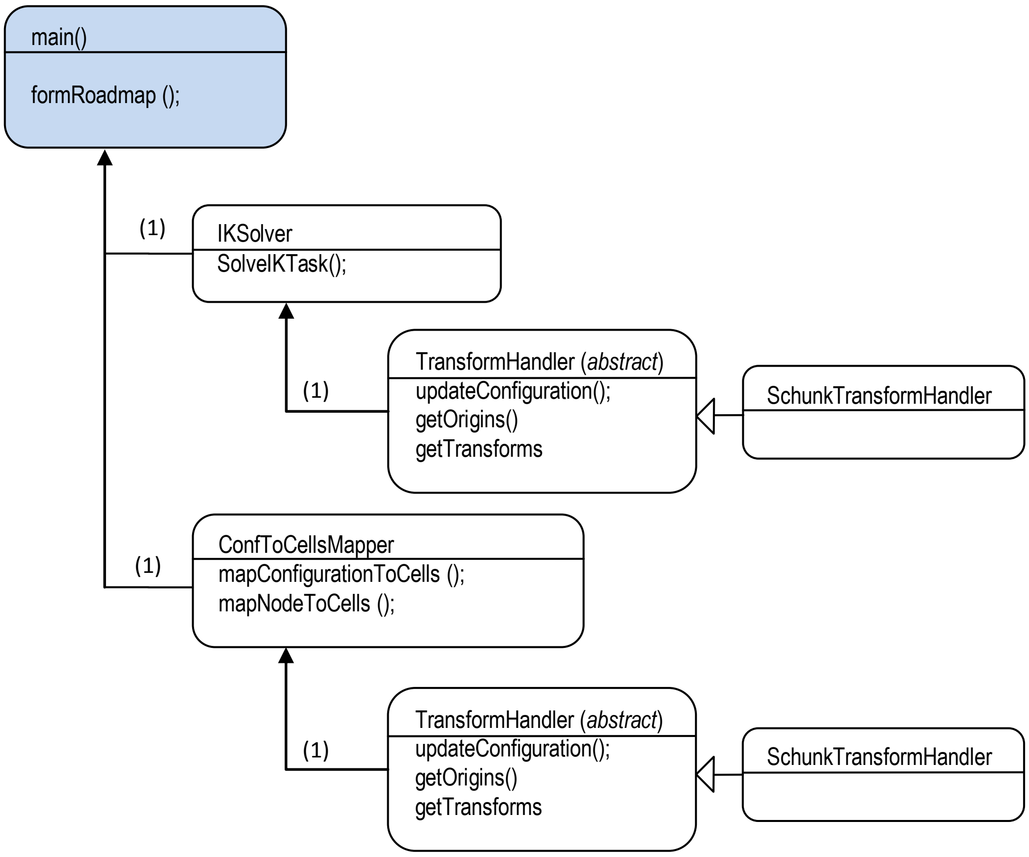


Fig. 33. Simplified class diagram of the offline part of the framework. Black-headed arrows represent “contain” relationship, white-headed arrows represent “derived from” relationship. The blue-coloured classes run as separate threads.

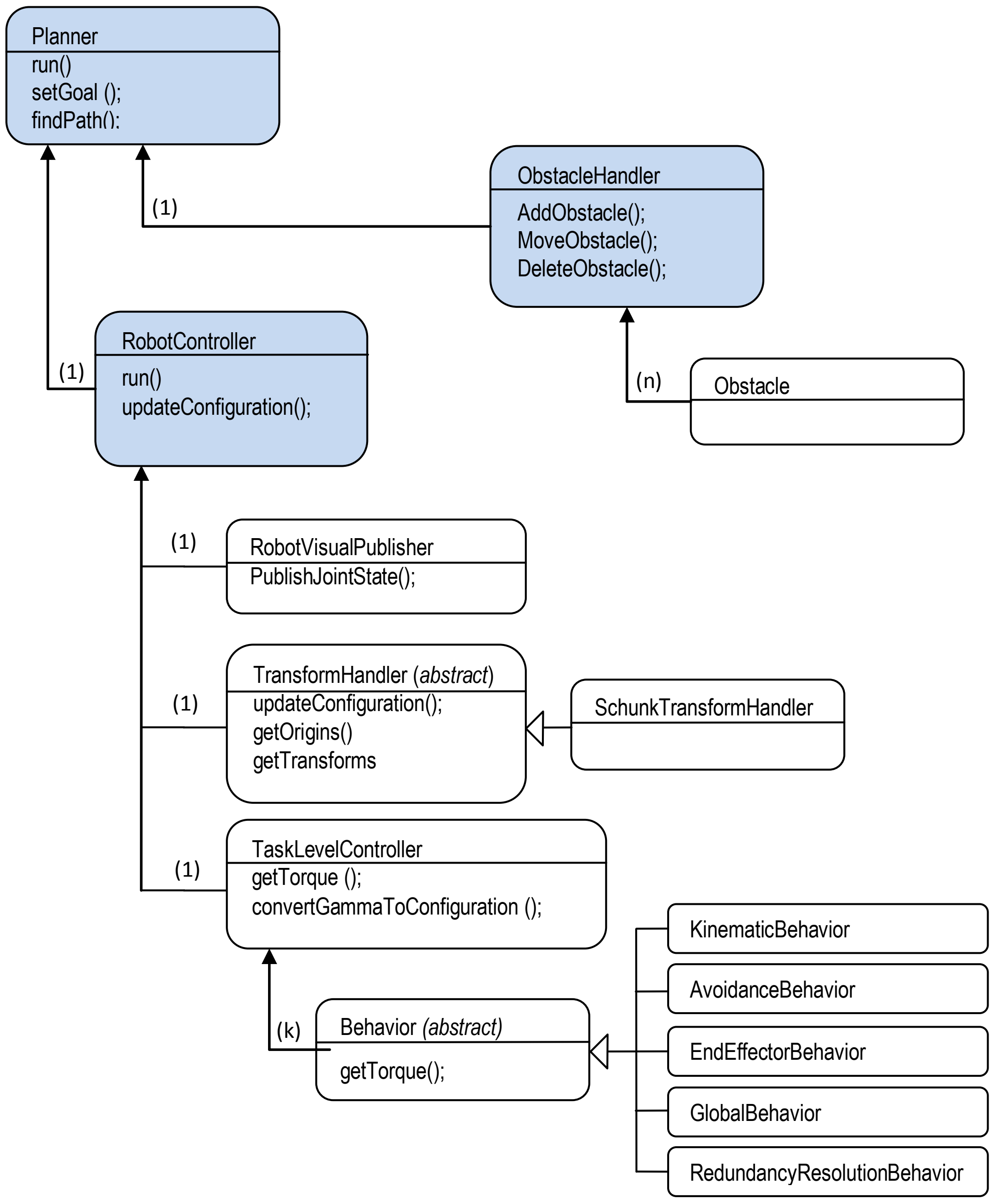


Fig. 34. Simplified class diagram of the online part of the framework. Black-headed arrows represent "contain" relationship, white-headed arrows represent "derived from" relationship. The blue-coloured classes run as separate threads. Behaviours represent the tasks for the task-level controller.

## 4.2. Representation of robot model

The model of the robot is stored in the *TransformHandler* class. Basically, it is a set of coordinate transforms that represent positions of the robot joints. The coordinate transforms for this and all other classes are implemented with the help of Bullet Physics Engine [**52**]. All other classes refer to *TransformHandler* when they need some data about the current state of the robot, e.g. position of origins of joint coordinate frames, orientation of joint axes or coordinate transforms from inertial frame to one of joint's frames.

The *TransformHandler* class is abstract and the exact coordinate transforms for a specific robot are performed in the derived classes, such as *SchunkTransformHandler*. This is done to provide a general interface to robot's model that can be used when applying the framework to another model of the robot. *SchunkTransformHandler* class additionally stores such data as joint limits and mass parameters of the links.

## 4.3. Roadmap representation, graph search and data storing

For effective handling of graphs the Boost Graph Library (BGL) [**53**] was used. BGL includes a set of template classes that allow fast creation and maintenance of graphs of various types as well as templates for graph-exploring algorithms, such as sorting of nodes, path search etc.

The roadmap created in the current project is an instance of bidirectional *adjacency_list* class, where each vertex contains a list of outgoing edges (Fig. 35).

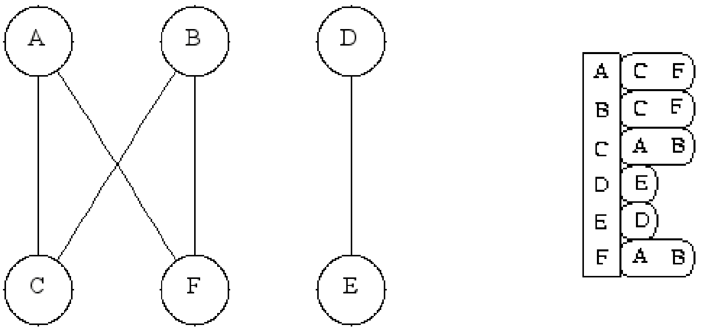


Fig. 35. Adjacency list representation of an undirected graph. [**53**]

The A* graph search algorithm is also implemented with BGL. To customize it for the specific needs a concept of *visitor* functions is used in BGL. Visitors are called whenever a certain point in the algorithm is reached. Out of eight visitors accessible for A* search the following ones were used in this project:

- **`examine_vertex`** – called when a new vertex is reached for the first time during the search. At this moment the vertex is tested for task consistency.

- **`examine_edge`** – called for all edges outgoing from the node checked in **`examine_vertex`** function. In this function if the node was task-inconsistent the weights of the edges are raised to some big value $N_{big}$ (the value is more than the number of nodes in the graph), so that they become an undesirable choice for being included in the path.
- **`finish_vertex`** – called when the goal vertex has been reached. At this point an exception is thrown to identify the end of search. When the exception is caught, we check the path cost that is the sum of edge weights when going from start vertex to the goal one. If the path cost is higher than $N_{big}$, it means that one of task-inconsistent nodes was included in the path or, in other words, that no task-consistent path has been found. Thus in this case the graph search is considered as unsuccessful. Otherwise, the path found is stored by the planner. After each graph search the edge weights changed in **`examine_edge`** function need to be restored to their initial values, so that they do not affect next search attempts.

One more important issue is how to store and load the created roadmap and its mapping to workspace cells. This is done using the Boost Serialization library [**54**]. This library provides methods to serialize a linked data structure and to store it into a file, from where it can later be loaded and deserialized. The Boost Serialization library serializes an object by serializing all its fields, recursively following pointers to other objects. It detects an object that has already been serialized and, thus, can handle circular pointer structures like our roadmap.

The size of the file containing roadmap (grid resolution 5 cm) is 5 MB. The size of the file containing mapping of configurations to the work cells (workspace grid resolution 3 cm) is 270 MB.

## 4.4. *RobotController* class

The *RobotController* class is just a container that stores the task-level controller and the model of the robot and provides communication between them and the planner. It runs in a separate thread and its main goal is just asking for new torque vector from the task-level controller and updating the model according to it (Fig. 36). It also updates the visual model as described in section 4.5.

```
while(true)
{
  newGamma = myTLC->getTorque();
  newConfiguration = myTLC->
     convertGammaToConfiguration(newGamma);
  myTransformhandler->setConfiguration(newConfiguration);
  rvizPublisher->PublishJointState(newConfiguration);
  loop_rate.sleep();
}
```

Fig. 36. Simplified main loop of the *RobotController* class.

## 4.5. Implementation of task-level controller

All the robot's tasks (or *behaviours*) have a lot of common functionality, therefore it is reasonable to have a general *Behaviour* class implementing it and derive individual behaviours from this class. The general class is abstract and it has the structure given in Fig. 37.

```
class Behaviour
{
 public:
   Behaviour();
   ~Behaviour();
   virtual const forceVector& getTorque()
   {
     calculateReferenceAcceleration();
     calculateNextTaskNullspace();
     TransformRefAccelerationToTorques();
     return myTorques;
   }

 protected:
   virtual void calculateReferenceAcceleration() = 0;
   Eigen::MatrixXd calculateOperationPointJacobian(uint OPIndex);
   void calculateNextTaskNullspace();
   void TransformRefAccelerationToTorques();
   forceVector myTorques;
};
```

Fig. 37. The most important members of the abstract *Behaviour* class.

Beside constructor and destructor, the main externally accessible function of the class is ***getTorque***(), which returns $\Gamma_{k|prec(k)}$ generated by the task. This process is divided into three subfunctions:

- ***calculateReferenceAcceleration*** calculates $\ddot{x}_{des}$. This function is purely virtual as each behaviour has its own rules how to do it. This function also computes the task Jacobian. The exact functionality of this function for different tasks corresponds to section 3.5.2.

- ***calculateNextTaskNullspace*** takes the nullspace resulting from the higher-priority tasks, calculates $\Lambda_{k|prec(k)}$ and the resulting nullspace.
- ***TransformRefAccelerationToTorques*** computes $F_{k|prec(k)}$ from $\ddot{x}_{des}$ and $\Lambda_{k|prec(k)}$ and transforms it into $\Gamma_{k|prec(k)}$.

The tasks acting in configuration space compute $\ddot{q}_{des}$ instead of $\ddot{x}_{des}$ and thus do not require the last step. Therefore, ***getTorque***() function is also virtual and if required, it can be reduced only to ***calculateReferenceAcceleration*** and ***calculateNextTaskNullspace*** functions. Transformation of $\ddot{q}_{des}$ into $\Gamma_{k|prec(k)}$ can be done after the calculation of reference acceleration in the first of the two functions.

The *Behaviour* class also has ***calculateOperationPointJacobian*** function. It has one argument that is the number of the coordinate frame whose origin is an operation point for the task. The function uses coordinates of this origin and calculates its full Jacobian according to algorithm from section 3.4.2. The resulting matrix is used to define the task Jacobian of behaviour.

All matrix calculation including inversion and SVD factorization are implemented with the help of Eigen C++ libraries – a comprehensive set of tools for linear algebra calculations [**55**].

With architecture of behaviours described above the class implementing the task-level controller has a simple structure (see Fig. 38).

```
class TaskLevelController
{
  public:
    TaskLevelController();
    ~TaskLevelController();
    forceVector getTorque();
    Configuration convertGammaToConfiguration(forceVector
gamma);
  protected:
    void computeInertiaMatrices();
    std::list<Behaviour*> myBehaviours;
    Eigen::MatrixXd H;
    Eigen::MatrixXd H_inverse;
    Eigen::MatrixXd N;
```

Fig. 38. The most important members of *TaskLevelController* class.

The dynamic list ***myBehaviours*** stores all behaviours the task-level controller has to maintain. The order of the behaviours in the list corresponds to their priorities: the closer it is to the beginning of the list, the higher priority it has. Which behaviours are included is defined in the owning *RobotController* class.

The ***getTorque*** function incorporates the main functionality of the class. It iterates through all the behaviours and sums up the generalized force vectors $\Gamma_{k|prec(k)}$ consistent with task hierarchy. The resulting generalized force vector is returned to the caller, which is usually an instance of the *RobotController* class. Throughout interaction with the behaviours the matrix ***N*** stores intermediate value of the behaviours' nullspace.

The function ***convertGammaToConfiguration*** implements the robot simulation model described in chapter 3.6. It is called for simulating robot movement in accordance with some torque vector calculated before. It also updates joint velocities vector, which is later used in computation of torques of behaviours.

The function ***computeInertiaMatrices*** computes joint-space inertia matrix and its inverse according to Composite-Rigid-Body algorithm (see chapter 3.4.3). It exploits three classes representing spatial 6D vector, spatial inertia and spatial coordinate transform to implement the spatial algebra operations. All computations with the spatial entities are, of course, performed in compact form.

## 4.6. Visualization of the model

For visualisation of the robot we use *rviz* module of the Robot Operating System (ROS) [**56**]. To explain the visualization process some introduction into ROS messaging mechanism needs to be presented.

### 4.6.1. ROS messaging mechanism

ROS is an open-source, meta-operating system for robots. It provides services one would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management. It also provides tools and libraries for obtaining, building, writing, and running code, also across multiple computers. ROS works on top of UNIX-based operating systems and is partially compatible with Windows. Full support of C++ and Python is provided.

One of the most powerful and easy-to-use services of ROS is communication between processes. More specifically, communication is performed between ROS *nodes* – some executables running as separate processes on one or several computers. Two mechanism of message passing are used: request/response and publish/subscribe. The latter mechanism is used in this project and, therefore, is presented in details.

When a ROS node starts, it registers its name at the Master node, which is the core of ROS. Then a node announces that it is going to publish (or subscribe) to a specific topic with a predefined name. If another node announces subscription (or publishing) to the same topic, the Master node informs each node about another and the nodes may start communication. The message passing goes not through the Master node, but through peer-to-peer connection between nodes; the Master node just helps publishers and subscribers to find each other.

Each topic has a predefined format of permissible messages and the format must be preserved by the publishers. The communication process is completely asynchronous. What is more, a topic may have several publishers and/or subscribers (Fig. 39). In this case whichever publisher posts a message, all subscribers receive it and they cannot distinguish who they receive it from.

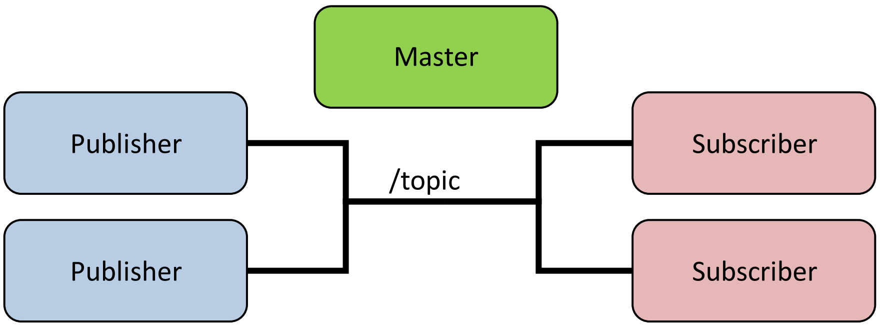


Fig. 39. Schematic representation of publish/subscribe communication.

#### 4.6.2. Communication with *rviz* module

As previously mentioned *rviz* is a module of ROS. It is used to visualize robots and their environments. It updates picture in accordance with messages published on some predefined topics. The planner itself works as a ROS node which updates configuration of visualization robot model and publishes some *markers* (objects for simple geometrical shapes) to represent obstacles and some additional objects. *Rviz* also contains convenient tools to adjust the visualized scene, such as moving and rotating the viewpoint, turning on and off visualization of some objects etc.

As ROS is specifically design to handle robotic systems, the visualization model can be easily constructed with a special version of XML language - Unified Robot Description Format (URDF). The model consists of a set of bodies representing robot links and a set of coordinate transforms for robot’s joints. The model used in the current project was designed by SINTEF for their internal projects.

The visualization robot model defined in an *.urdf* file is first loaded to ROS Parameter server. When *rviz* starts, it loads the model and then it requires only joint angles published by the planner to update the image.

# 5. Simulation and results

To verify the functionality of the proposed framework it was tested under various conditions. Each test scenarios (experiments) were carried out by simulating behaviour of the 7-DOF Schunk manipulator (Fig. 1) controlled by the designed framework. The simulation model or the robot is presented in the Chapter 3.6.

For each experiment a global task (i.e. moving the end effector from one point to another) and end effector orientation task can be specified. Moreover, it is possible to add obstacles to each of the test scenarios. At the current stage of the project, an obstacle can be represented as a rectangular or spherical object. Obstacle movement is modelled by discrete changing of its position (no real notion of velocity is used).

With the simulations we want to demonstrate the capabilities of the planner (graph search, replanning, failure detection) and the task-level controller (how separate behaviours perform and how the prioritization mechanism works). For this goal a set of various tasks has been designed for experiments 1 to 8. The complexity of tasks gradually increases starting from a simple point-to-point task and finishing with point-to-point movement under orientation constraints in the presence of obstacles. The experiments 9 and 10 reveal the limitations imposed on the framework's functionality by simplifications and decisions made during design process.

It is necessary to say, that in all experiments the structure of the framework remains the same, e.g. the task-level controller has the same setup as described in Chapter 0, all behaviours are always on etc. Only the tasks and/or obstacles change from one experiment to another.

## 5.1. Timing characteristics

The simulation was performed on computer with Intel Core i5 (2,26 GHz) processor with 4 GB of RAM.

The times required to perform some of the main processes of the proposed framework are summarized in Table 4.

| Process | Average time, seconds |
|---|---|
| Calculations during one control cycle (including computation of the torque vector by the task-level controller and movement simulation) | 0,003-0,006 |
| Path search (successful) | 0,014-0,027 |
| Path search (unsuccessful) | 0,047 |
| Obstacle movement (the big spherical obstacle from the Experiment 5) | 0,025-0,027 |

Table 4. Time required to perform the main processes of the proposed framework.

## 5.2. Experiment 1. Simple point-to-point task

**Goal of the experiment:** To demonstrate basic functionality of the planner and the task-level controller

**Scenario:** The robot is set to move its end effector from one point to another. No task constraints are specified. The environment is obstacle-free.

**Results:** The course of the scenario execution is shown in Fig. 40. The graphs demonstrating the change of the position of the end effector is shown in Fig. 41.

The framework easily deals with this simple task.

When no task constraints are specified, any node that contains at least one obstacle-free configuration can be included into the path. The graph search algorithm finds the shortest path leading to the desired position of the end effector and the task-level controller is able to guide the robot from one node to another. The graphs on Fig. 41 show that the end effector moves towards the goal position located at the point A(0.2;0.4;0.2) without big deviations or oscillations.

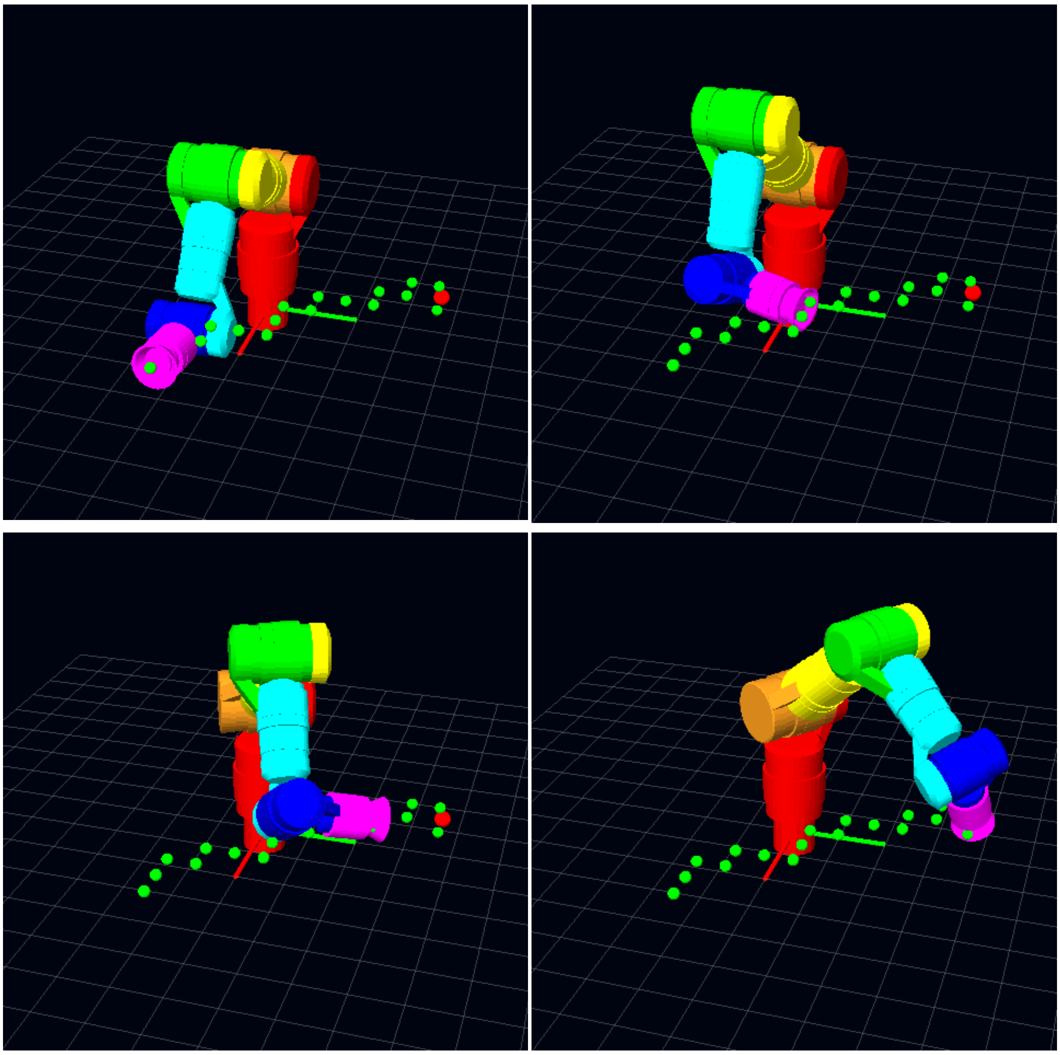


Fig. 40. The course of the experiment 1. The red point is the goal, the green points mark the path found by planner.

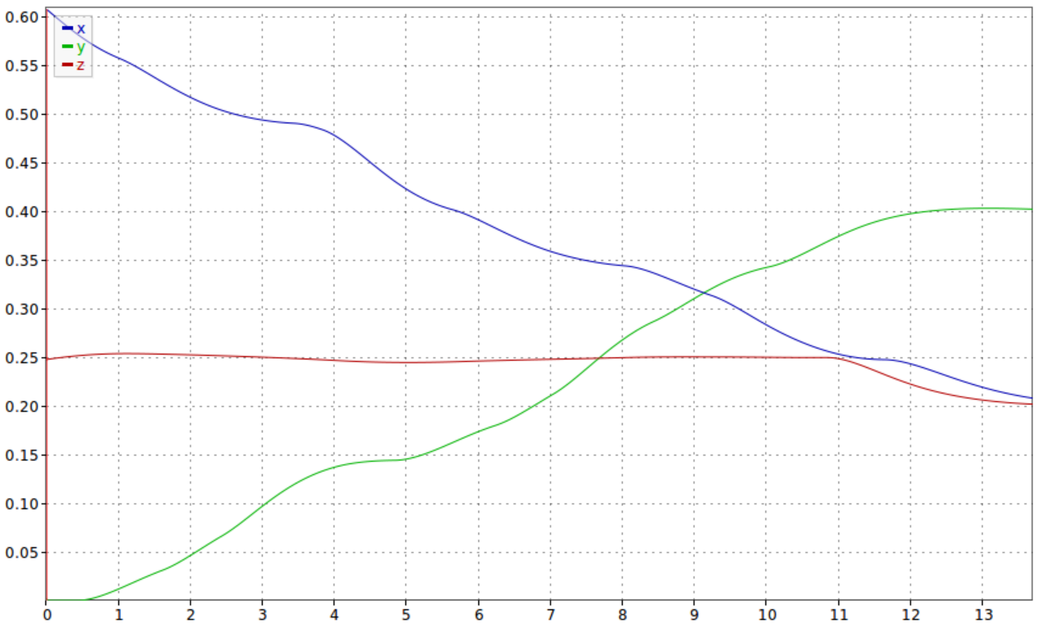


Fig. 41. The change of x, y and z coordinates of the end effector in time during the experiment 1. The graphs were obtained with *rosbag* tool – one of multiple ROS utilities.

### 5.3. Experiment 2. More complex point-to-point task

**Goal of the experiment:** Demonstration of failure detection and joint limit avoidance.

**Scenario:** The robot is set to move its end effector from one point to another. The starting point is located in front of the robot while the goal point is located behind it. No task constraints are specified. The environment is obstacle-free.

**Results:** The course of scenario execution is shown in Fig. 42. A video recording of the experiment can be found in the digital attachments and on http://youtu.be/w4Als_PjKsY.

During execution of this scenario several replanning attempts were needed (pay attention to different paths in Fig. 42 b-d, f). Replanning occurs when the planner detects that no progress towards the goal waypoint has been made during a certain amount of time. In Fig. 42 b this situation is caused by reaching the joint limits by the robot. The task-level controller cannot find a solution to this situation and an external assistance (i.e. replanning) is needed. At first, the planner suggests a similar path that goes a bit lower than the original one, but it led to one more joint-limit issue (Fig. 42 c). The problem was finally solved when the planner suggested the path that goes to the side from the initial one (Fig. 42 d). The task-level controller then is able to find a solution and follow it avoiding collision with the base of the robot.

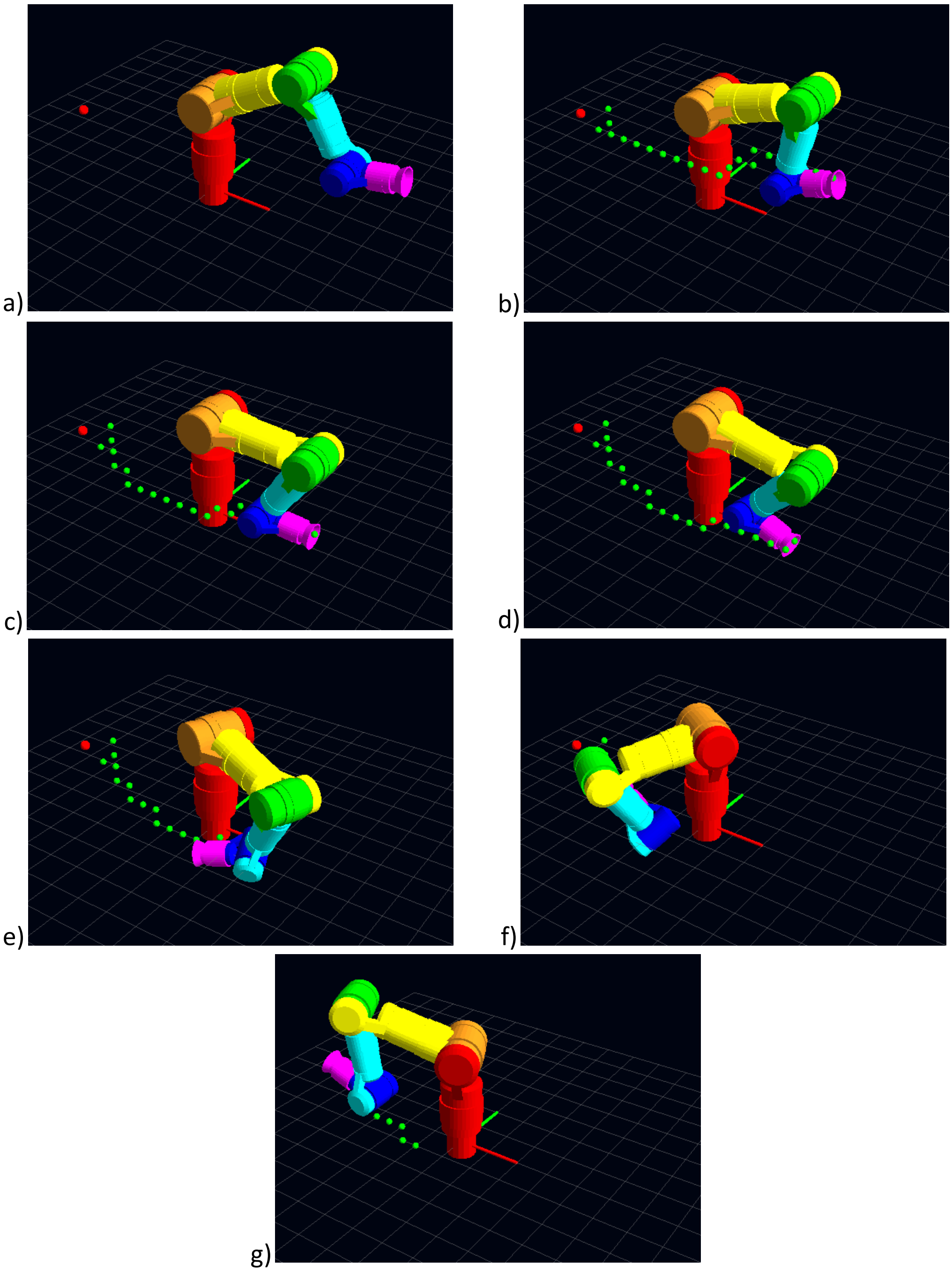


Fig. 42. The course of the experiment 2. The red point is the goal, the green points mark the path found by planner.

Although no task constraints are explicitly included in this experiment, the simulation demonstrates how positional task constraints can be integrated into the framework. As can be seen in Fig. 42 b-e, the path found during the graph search goes around the base of the manipulator. The reason for this is that the nodes in this region (though they exist in the roadmap and store some configurations computed during preplanning) are considered as task-inconsistent for any task and thus the planner avoids including them into the path.

## 5.4. Experiment 3. A point-to-point task with feasible task constraints

**Goal of the experiment:** Demonstration of path following with preserving orientation task constraints (aiming with the end effector at a fixed point in space).

**Scenario:** The robot needs to move its end effector from one point to another and to try to keep such orientation of the end effector, that it is aimed towards a point fixed in 3D space throughout the movement. The environment is obstacle-free.

**Results:** The course of scenario execution is shown in Fig. 43. A video recording of the experiment can be found in the digital attachments and on http://youtu.be/raPfsC3mu1I.

From the figures it is clearly seen that all tasks are successfully completed. The scenario execution goes in two phases. The initial configuration of the robot (Fig. 43 a) is task-inconsistent and in the first phase the manipulator comes to a task-consistent configuration. During this time it already starts moving towards the goal point (Fig. 43 b). After the end effector reached its desired orientation, the rest of the path is followed in task-consistent manner (Fig. 43 c-f).

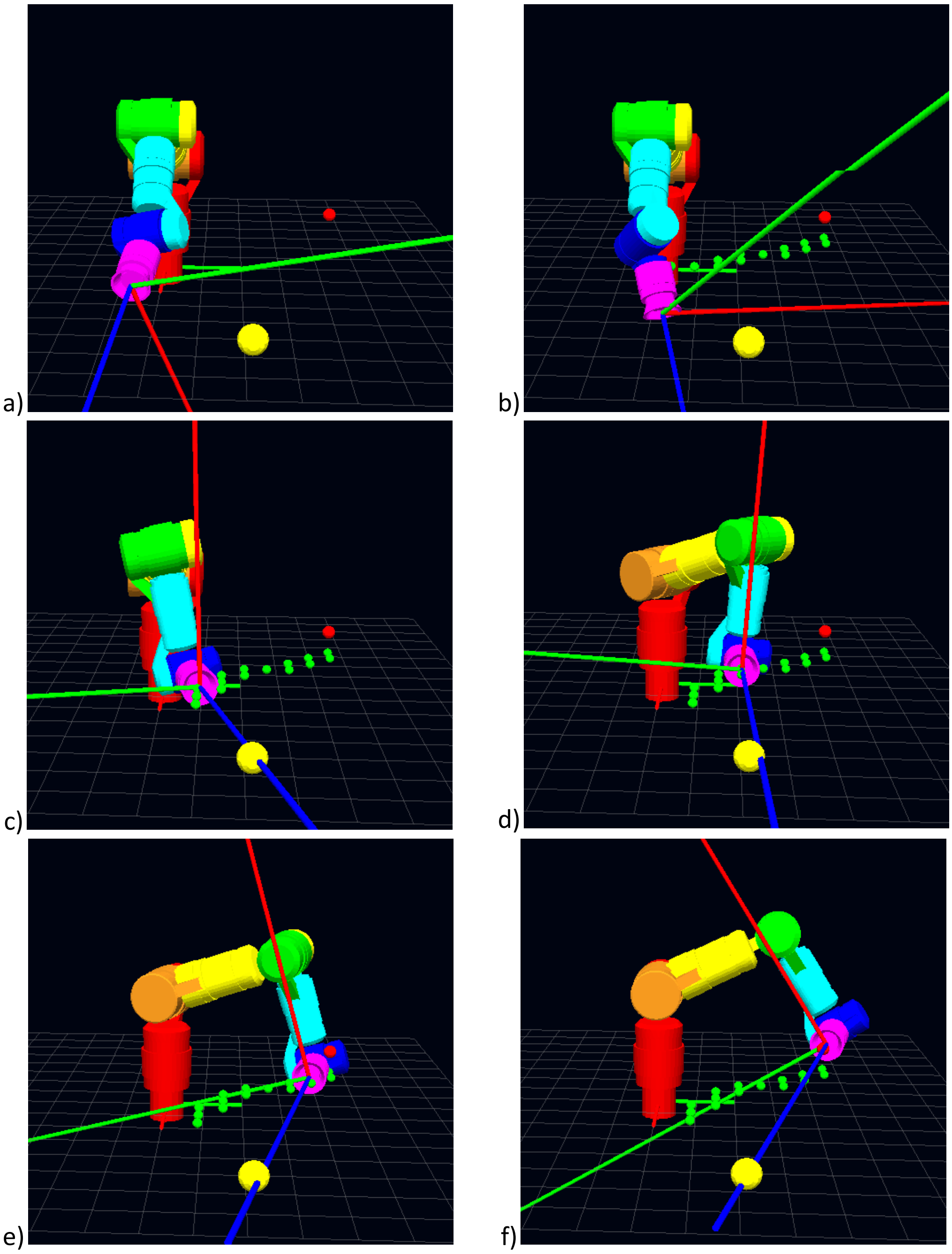


Fig. 43. The course of the experiment 3. The red point is the goal, the green points mark the path found by planner, the yellow sphere (radius 2,5 cm) marks the point which the end effector should aim at. The coordinate frame attached to the end effector is shown so the following the task constraints is clearly seen (X-axis is red, Y-axis is green, Z-axis is blue). It may seem that the path waypoints go in pairs in the figures. This is just a visual effect; the path goes directly to the goal point and no mirroring or doubling of the waypoints occurs.

## 5.5. Experiment 4. A point-to-point task with unfeasible task constraints

**Goal of the experiment:** Testing detection of unfeasible task constraints.

**Scenario:** The robot has to repeat point-to-point motion as in the previous experiment but with different task-constraints. It should keep the end effector oriented in the positive direction of the Z-axis (upright orientation) during the task execution. The environment is obstacle-free.

**Results:** The setup of the experiment is shown in Fig. 44.

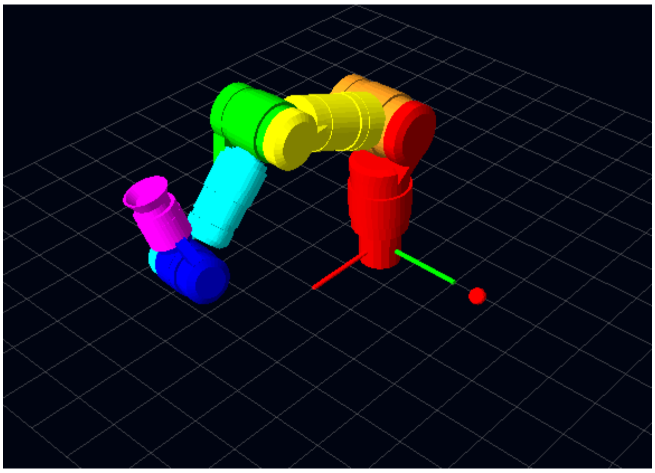


Fig. 44. The setup of the experiment 4. The red point is the goal for the end effector.

Scenario execution again goes in two steps: finding an initial task-consistent configuration and the following the path in task-consistent manner. However, neither of the steps is successful and the planner cannot find a feasible path.

The failure of the first step is caused by the structure of the end-effector behaviour ($\varphi_{task}$) of the task-level controller. It affects only the joints 5 and 6, but the latter one reaches its limit in the configuration shown in Fig. 44. The other joints are not affected by the behaviour and a task-consistent configuration cannot be reached.

In the current project task-inconsistency of the initial configuration is not considered as a failure and the planner still tries to find a feasible task-consistent path to the goal. However, as the initial position of the end effector cannot be mapped to any task-consistent node, no feasible path can be found. In this case we can say that it is impossible to preserve the orientation constraints along the path with the limitations imposed on possible configurations in the Chapter 3.2.2 (i.e. $q_3 = 0, q_4 > 0$). The framework cannot solve this task and this meets its expected behaviour.

## 5.6. Experiment 5. Obstacle avoidance

**Goal of the experiment:** Demonstration of online reactive obstacle avoidance.

**Scenario:** The robot has neither point-to-point task, nor orientation constraints. A spherical-shaped obstacle moves in such way that it may collide with the robot.

**Results:** The course of scenario execution is shown in Fig. 45. A video recording of the experiment can be found in the digital attachments and on http://youtu.be/sfBvIBbMWKg.

As no task is specified for the robot, the planner is inactive and obstacle avoidance is performed only by the means of task-level controller, namely by the collision-avoiding behaviour $\varphi_{collision}$. The safety distance where $\varphi_{collision}$ activates is 3 cm.

The obstacle movement is modelled by changing its position by 5 cm every half a second. Thus we can say that it moves with the speed of 0,1 m/s. However, as the movement is discrete, it may come too close to the robot at some points of time (as in Fig. 45 b).

The figures above clearly demonstrate that the obstacle avoidance task may effectively prevent collisions with obstacles moving in the robot's workspace.

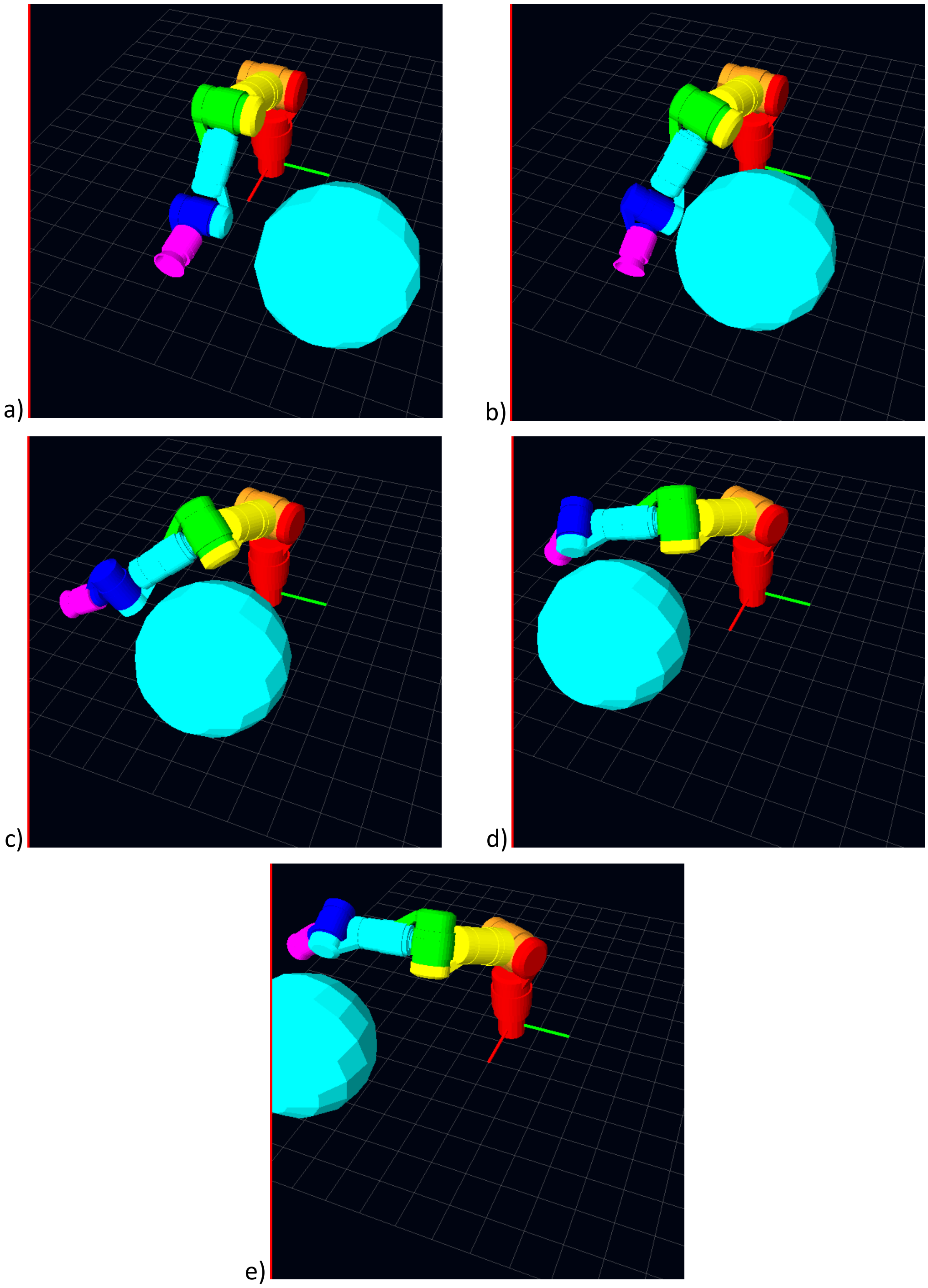


Fig. 45. The course of the experiment 5.

## 5.7. Experiment 6. A point-to-point task with task constraints in presence of obstacles

**Goal of the experiment:** This experiment demonstrates the main functionality of the proposed framework and tests task-consistent movement of the robot in presence of obstacles.

**Scenario:** The same, as in experiment 4, but in presence of a static obstacle obstructing the straight path.

**Results:** The course of scenario execution is shown in Fig. 46. A video recording of the experiment can be found in the digital attachments and on http://youtu.be/UYl4MN-SeN8.

Fig. 46 demonstrates that the test scenario is successfully completed. Compared to the path used in Experiment 4, the robot made a big detour around the obstacle. At some point (Fig. 46 d) the orientation task constraints were violated during the path execution. However, this happened not because of inappropriate planning. The real cause for it is that the behaviour controlling end effector's orientation was too slow to react to the change of end effector's position due to other behaviours. In the current implementation task violation is not considered as a failure and the scenario execution continued. The correct orientation was reached again soon (Fig. 46 e) and the rest of the path was executed in task-consistent manner.

For crucial tasks violation of task constraints may be considered as a failure. In this case the framework should stop following the path, wait until the task constraints are satisfied and continue after that. The way how the framework treats violation of task constraints depends only on designer's decision.

Note also, that all degrees of freedom (including the joint 2 (yellow link in the figures)) are used and the soft constraint imposed by low-priority task $\varphi_{posture}$ is violated at some part of the path. However it is satisfied wherever it is possible.

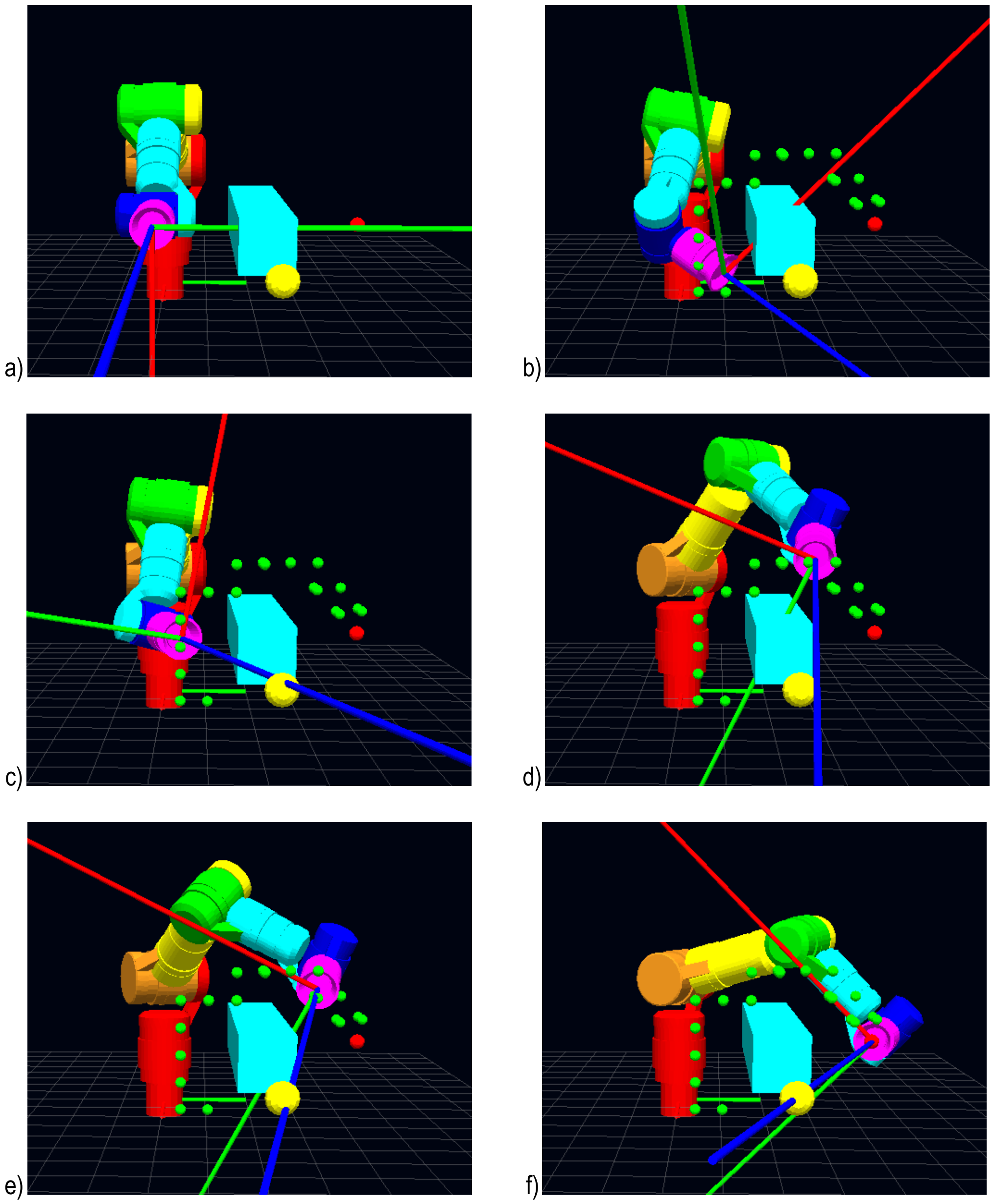


Fig. 46. The course of the experiment 6. The red point is the goal, the green points mark the path found by planner, the yellow sphere (radius 2,5 cm) marks the point which the end effector should aim at. The coordinate frame attached to the end effector is shown so the following the task constraints is clearly seen (X-axis is red, Y-axis is green, Z-axis is blue).

## 5.8. Experiment 7. A point-to-point movement with fixed orientation in presence of obstacles

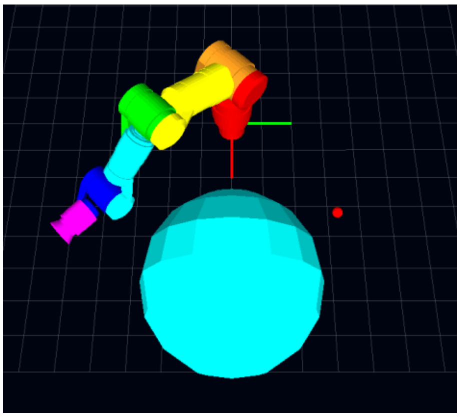
**Goal of the experiment:** This experiment demonstrates how the framework copes with the second type of task constraints, when the end effector should preserve fixed orientation.

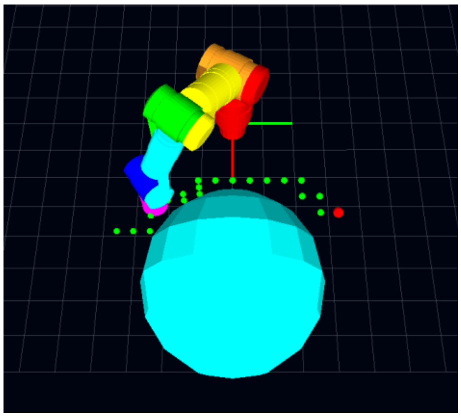
**Scenario:** The robot needs to move its end effector from one point to another while keeping the end effector oriented in the negative direction of the Z-axis (downright orientation) along the way. A static spherical obstacle obstructs the direct path.

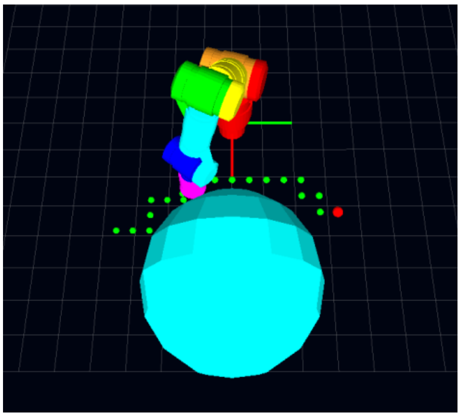
**Results:** The course of scenario execution is shown in Fig. 47. A video recording of the experiment can be found in the digital attachments and on http://youtu.be/aN2MGxaN0YA.

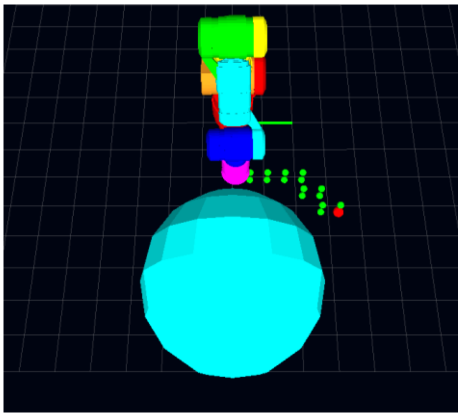
The framework was capable to ensure successful execution of the test scenario. At first, a task-consistent initial configuration was found, and the rest of the path was followed in task-consistent manner. A replanning event was needed in the middle of the path (Fig. 47 d) as obstacle avoidance behaviour contradicted the task following the path. As $\varphi_{collision}$ has higher priority, no progress towards the waypoint was made during some time, which triggered a replanning attempt.

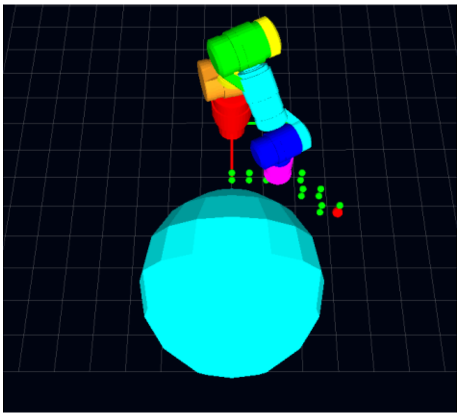
This experiment also demonstrates that representation of obstacles as a set of occupied workspace cells is more effective than approximating them with a bounding box (as it is done in *elastic roadmap* framework). In the current setup, the robot moves diagonally in the last part of the path which would be impossible if the sphere was approximated with one big cube.

a) 

b) 

c) 

d) 

e) 

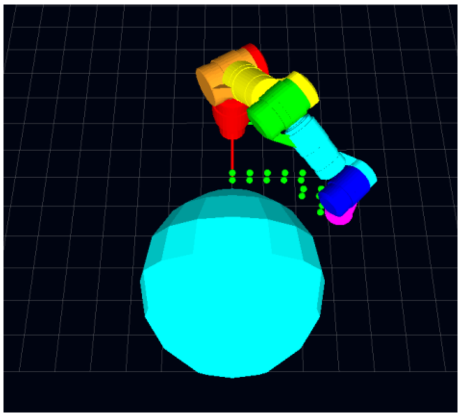
f) 

Fig. 47. The course of the experiment 7. The red point is the goal, the green points mark the path found by planner (in figures (d), (e) and (f) the real path is the one, going above; the lower one is shown due to an error in visualization).

## 5.9. Experiment 8. A point-to-point task in presence of moving obstacles

**Goal of the experiment:** To demonstrate replanning capabilities in presence of moving obstacles.

**Scenario:** The robot needs to move its end effector from one point to another. A spherical obstacle moves in such way that it obstructs the direct path that was unobstructed at the beginning. No orientation task constraints are specified.

**Results:** The course of scenario execution is shown in Fig. 48. A video recording of the experiment can be found in the digital attachments and on http://youtu.be/Jj_2VnOCWcA.

The spherical obstacle continuously obstructs the paths found by the planner (Fig. 48 a-d). The framework quickly responses to the changes in the environment and suggests new paths. During this period a few replanning events needed due to reaching the joint limits (same as in Experiment 2). After the obstacle finally stops (Fig. 48 e), the robot continues execution of the last found path. Several replanning attempts needed to finish the path and avoid collision with the obstacle (Fig. 48 f,g).

This experiment shows interaction between the planner and the task-level controller. When the controller gets stuck due to reaching joint limits or trying to go around an obstacle, the planner finds an alternative path that possibly can guide the task-level controller out of the inappropriate state.

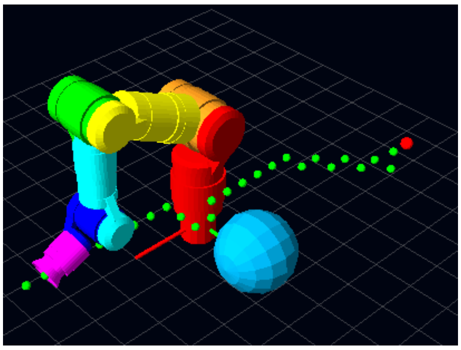
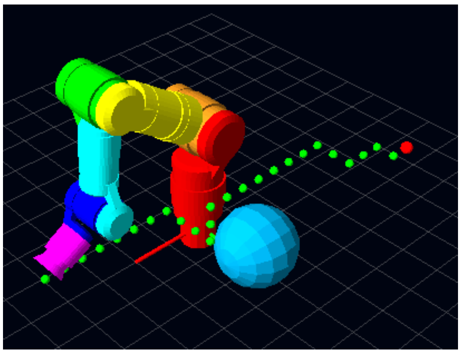
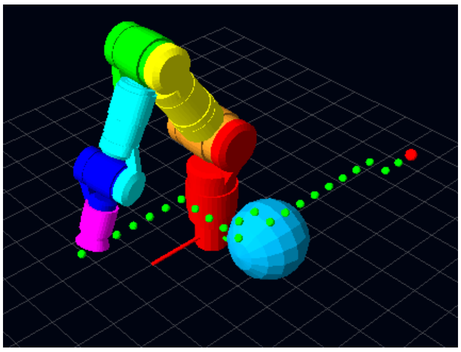
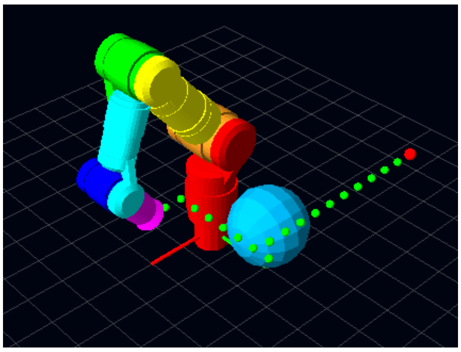
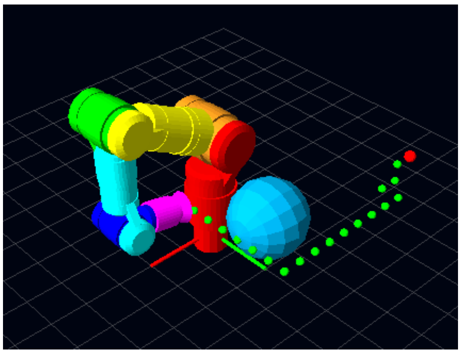
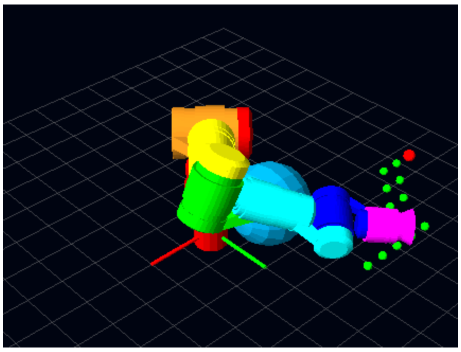
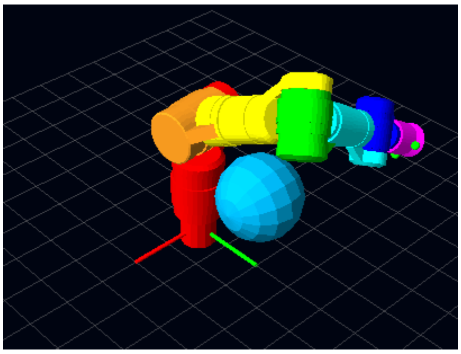
a)

b)

c)

d)

e)

f)

g)


Fig. 48. The course of the experiment 8. The red point is the goal, the green points mark the path found by planner.

## 5.10. Experiment 9. Framework limitations

**Goal of the experiment:** To demonstrate limitations imposed by the structure of the suggested framework.

**Scenario:** The robot needs to move its end effector from one point to another. The goal point is located close to the bottom side of an obstacle. The orientation task is to point in the positive direction of X-axis.

**Results:** The setup of the experiment is shown in Fig. 49.

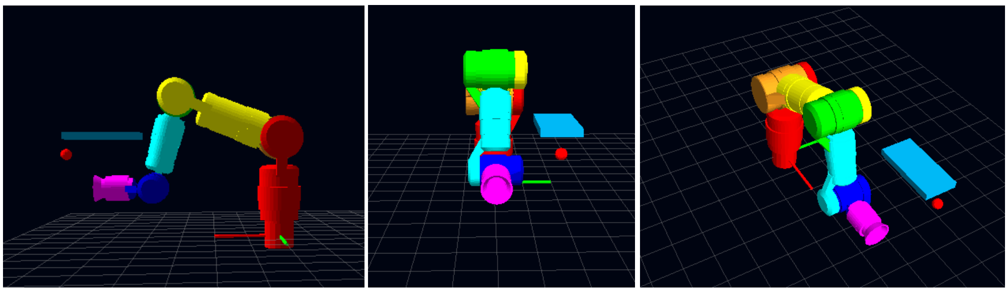


Fig. 49. The setup of the experiment 9. The red point is the goal for the end effector.

The framework cannot find a solution in the presented scenario although it is obvious that a feasible solution exist if the 4-th joint is assigned a negative value and the goal point can be reached “from the bottom”). The false negative occurs due to assumptions made during preprocessing stage (i.e. constraints on the joints 3 and 4). This problem will be discussed in more details in the next chapter.

## 5.11. Experiment 10. Graph search limitations

**Goal of the experiment:** To demonstrate limitations imposed by the structure of the suggested framework.

**Scenario:** The robot needs to move its end effector from one point to another and try to keep such orientation of the end effector that it points at the same point in 3D space throughout movement. The starting configuration is different from the one used in most of previous experiments. The environment is obstacle-free.

**Results:** The setup of the experiment is shown in Fig. 50.

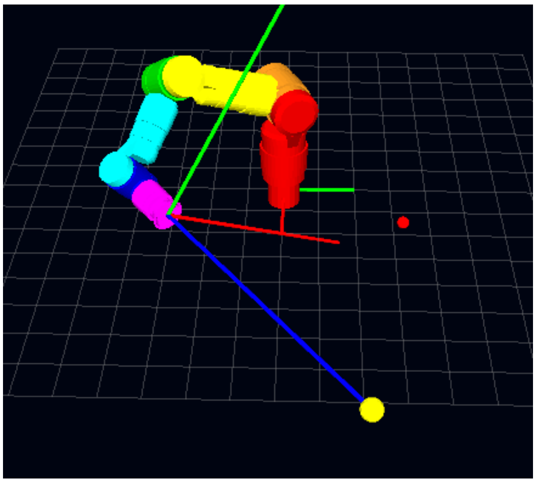


Fig. 50. The setup of the experiment 10. The red point is the goal for the end effector. The yellow sphere (radius 2,5 cm) marks the point which the end effector should aim at. The coordinate frame attached to the end effector is shown so the following the task constraints is clearly seen (X-axis is red, Y-axis is green, Z-axis is blue).

The initial configuration satisfies task constraints and from the previous experiment we can suppose, that a task-consistent path exists. However, the framework cannot find it. The reason for this is that the initial position of the end effector is considered as task-inconsistent by the planner. In the current setup, to be task-consistent the node corresponding to the initial configuration should contain at least two configurations: one corresponding to end effector oriented in the direction of X-axis (red line at the robot's base) and one corresponding to end effector oriented in the direction of Y-axis (green line at the robot's base). This comes from the algorithm which is used for task-consistency check of nodes (see chapter 3.3.2). However, due to the assumptions made for the offline-stage inverse kinematics solver (chapter 3.2.2), no configuration corresponding to the Y-axis has been found. Therefore, the initial configuration cannot be mapped to a task-consistent node and the framework fails to find an appropriate solution.

# 6. Discussion

The goal of this project was development of a framework that could provide task-consistent movement of a manipulator in unstructured dynamic environment. In this chapter we will discuss how designed system corresponds to this goal and identify the main shortcomings of the proposed solution.

As it was demonstrated with simulations in the previous chapter, the overall performance of the proposed framework meets the requirements set at the beginning of the project. To be more specific, the system possesses the following features:

- It can handle both positional and orientation task constraints and considers them during path planning as well as while executing the path.
- Combination of a planner, based on *Dynamic roadmaps (DRM)*, and reactive obstacle avoidance ($\varphi_{collision}$) provide broad capabilities to prevent collisions with arbitrary moving obstacles of complex shape.
- The path planner is able to identify collision-free paths of the end effector that can be at least approximately followed in task-consistent manner.
- The task-level controller itself may solve simple local tasks.
- The guidance of the task-level controller by the path planner, that has a global overview of the workspace, allows solving of complex tasks and finding alternative solutions when the task-level controller gets stuck and cannot move towards next waypoint because of obstacles or joint limits.
- The behaviour $\varphi_{posture}$ provides coherence between online and offline stages.
- Fulfilment of real-time constraints imposed by dynamic environments is ensured by the fact that search space is bounded. Because of that the path search does not exceed some finite time. This time depends on size of the roadmap which is reduced significantly by storing six configurations in one node of the roadmap as this minimizes number of states within search space compared to classical DRM. The time needed for replanning determines system capabilities in the context of fast moving obstacles.
- From the implementation point of view the framework is reasonably structured and can be integrated into a more complex system with the help of ROS tools.

The proposed framework is based on *DRM* and *Elastic roadmaps* and, therefore, it inherits some features of both. A short comparison of the three frameworks is summarized in Table 5. The initial motivation for this project was to compensate the drawbacks of *Elastic roadmaps* by introducing offline stage. Namely, we wanted to improve obstacle representation and to reduce framework incompleteness. The first task is completed successfully whereas the latter issue requires more discussion.

| Criterion | Suggested framework | DRM | Elastic roadmaps |
|---|---|---|---|
| Handling dynamic environments | Yes | Yes | Yes |
| Task are specified in operational space | Yes | No | Yes |
| Handling task-constraints | Yes | No | Yes |
| Preprocessing stage | Yes | Yes | No |
| Obstacle representation | As a set of workspace cells | As a set of workspace cells | As bounding box |
| Method completeness | Incomplete | Probabilistically complete | Incomplete |
| Workspace size | Limited workspace | Limited workspace | Unlimited workspace |

Table 5. Comparison of the suggested solution with DRM and *Elastic roadmaps* frameworks.

## 6.1. Framework completeness

The main factor limiting the system's performance is its incompleteness. It can be divided in two parts. The first one is inherited from the *Elastic roadmaps*; the second part is a distinctive feature of the suggested design.

The authors of *Elastic roadmaps* discuss the incompleteness of their approach as follows: *"... The elastic roadmap framework is inherently incomplete and may fail even when a valid path exists... It does not possess any of the completeness properties of sampling-based planners. ... The elastic roadmap framework explicitly addresses task constraints and feedback requirements of a specific application and permits the execution of motion in dynamic environments under these constraints. It is able to do so precisely because it sacrifices completeness."* [**44**] The inherited incompleteness may reveal itself in situations when the task-level controller cannot move towards next waypoint (or *milestone* as denoted in *Elastic roadmaps*) and the guiding planner cannot find an alternative solution. Although such cases are not absolutely impossible in the proposed framework, they are more rare, than in *Elastic roadmap*. The reason for this is the thorough workspace exploration during the preprocessing stage employed by the framework presented in this report. Using this information during planning and trying to keep configurations close to the ones used in offline computations we increase the completeness of our method compared to *Elastic roadmaps*, where only a few samples of the workspace (obstacle-related milestones) are used for planning.

A possible improvement that would help to avoid situations that are hard to solve for the task-level controller is to include data about manipulability into the roadmap. For instance, each node would have a manipulability index that is calculated based on the number of configurations stored by in the node and manipulability indices of each one of them. The manipulability index of a configuration can be expressed as a distance measure of the configuration from the singular ones and it is usually computed from the eigenvalues of the manipulator's Jacobian matrix [**57**]. Using this data, the planner would include the nodes with high manipulability index into the path and the task-level controller would have more options to follow it locally while satisfying all constraints. For example, in the experiment 2 it would mean that the path would avoid nodes, where the manipulator is close to its joint limits, and would prefer the path where the end effector sweeps an arc from the initial to the goal point.

However, how the experiments 9 and 10 demonstrated, another source of incompleteness was introduced during the design of the suggested framework. It is the constraints used for inverse kinematics solving in the offline stage. In the current implementation we assumed that the angle of the 3rd joint is kept at zero and that the 4th joint may be assigned only positive values. These constraints bound the explored part of the 7D configuration space to one half of 6D plane, which of course reduces performance during path planning as many feasible task-consistent configurations are taken out of consideration.

One approach that could improve this situation is to create a second set of nodes that would correspond to configurations subjected to different constraints. For example, we could create a grid where the inverse kinematics would be solved with the same constraint for the 3rd joint, while the 4th joint would be assigned only negative values. This grid would correspond to all configurations approaching the goal point "from below" (e.g. to solve the task in experiment 9). Thus we would have two independent grids that need to be connected. Two nodes belonging to different grids and corresponding to the same position would contain similar configurations when the angle of the 4th joint is close to zero. Thus we can insert edges between such nodes and, therefore, connect the two grids and extend the part of configuration space explored in the offline stage. In this case we will have 4D roadmap: three dimensions refer to spatial location of the robot's end effector while the fourth dimension corresponds to the configuration type. Such roadmap would also require some changes in planner and the task-level controller. For the planner this would mean that the goal point can be mapped to two nodes. Thus the planner would have to find a path to each of them and then to choose the shorter one. For the task-level controller the main changes will be done inside $\varphi_{posture}$. It would have to change its objective depending on configuration type of the next node in the path. It could be also useful to disable $\varphi_{global}$ during transition between grids, so $\varphi_{posture}$ has more freedom to change the configuration.

In a similar way more grids can be built, for example for constraints like $q_3 = 90°, q_4 > 0$ and $q_3 = -90°, q_4 > 0$. However, this would lead to some negative consequences. Consider $n$ grids referring to different configuration types are used for roadmap construction. Then, the planning time would increase as the search space increases by $n$ times and several paths should be estimated. Additionally the file that stores roadmap mapping to workspace cells would increase by $n$ times. Even for one grid it has size of hundreds of megabytes (270 MB for the roadmap grid resolution of 5 cm and workspace cell size of 3 cm for the current manipulator) and it takes about 30 seconds to load it at the start of the framework. This would considerably increase the amount of time needed for obstacle handling affecting frequency characteristics of the system. This trade-off between framework completeness and timing parameters should be considered by system designers based on the requirements and types of possible tasks.

One more approach that would help solving the task in the experiment 10 is to store more samples of possible orientations of the end effector within one node. This would increase the probability that if the planner finds a path, it would be possible to maintain task constraints along it. The cost of this improvement is the increased amount of time needed for obstacle handling.

## 6.2. Other factors limiting performance of the framework

Apart from framework incompleteness, the following factors may limit its application in real life:

- Each behaviour within the task-level controller basically forms a potential field in operational or configuration space and the controller combines them according to their priorities. Thus, the task-level controller faces all problems inherent to general potential-field methods (see Section 2.3). Among them are the local minima problem (when the controller gets stuck at some point) and oscillations at the border where certain behaviour activates. For example, if $\varphi_{global}$ drives the robot towards an obstacle, $\varphi_{collision}$ would counteract this movement. But as soon as the robot is far enough from the obstacle, $\varphi_{collision}$ becomes inactive and $\varphi_{global}$ pulls the manipulator into the danger zone again. A similar situation occurs with behaviour avoiding joint limits ($\varphi_{kinematic}$). The guidance from the planner helps to resolve the situation, but the oscillations cannot be fully compensated.
- As no explicit trajectory is specified in the configuration space, the planner has no notion of time. Thus it is hard to include time-varying task constraints into the framework. For instance, if the task is to follow a moving object with

a videocamera, replanning is needed every time the object moves as its movement cannot be considered during the graph search.

- It is quite hard to map the suggested framework to robots with even bigger redundancy degree, than the manipulator considered in this project. In this case too many assumptions need to be done during the preprocessing stage and system incompleteness would increase greatly (although, the task-level controller would still provide at least partial functionality). On other hand, the method becomes almost exact when applied to non-redundant robots.
- The suggested framework does not consider velocities of obstacles, only their locations. Thus, to avoid fast moving obstacles the safety distance set for $\varphi_{collision}$ should be increased. Another way to handle such obstacles is to model them differently. For instance, a certain zone, whose size depend on the velocity of the obstacle, located in the direction of its movement should be also considered as occupied by this obstacle. In this case the robot would have more time and space to go from the obstacle’s path and thus to avoid the collision.

# 7. Future work

Beside the major improvements described in section 6.1 (storing manipulability data within nodes, augmenting roadmap with more nodes and storing more configurations within one node) the following directions can be considered for future work:

- Modification of $\varphi_{task}$, so that it works not in configuration, but in operational space;
- Computation of Coriolis and gravity terms for the robot's dynamics model;
- Friction modelling and compensation;
- Design of interfaces to robot sensors (for instance, a laser range scanner or videocameras to detect obstacles) using ROS communication tools;
- Design of a GUI for specification of manipulation tasks;
- Application of the framework to mobile manipulation with special consideration of interaction with the mobile platform.

# 8. Conclusion

The current project was devoted to design of a framework that could provide task-consistent movement of a 7-degree-of-freedom manipulator in unstructured dynamic environments. An extensive literature survey of motion planning techniques with a focus on sample-based and feedback planning, as well as planning in dynamic environments has been conducted. A solution based on two state-of-art approaches, namely *dynamic roadmaps* and *elastic roadmaps*, was proposed and implemented. The suggested framework uses information acquired during offline preprocessing stage to handle dynamic obstacles effectively and to guide task-consistent motion of the robot controlled by a task-level controller. During the project the offline preprocessor and online planner were designed from a scratch. General architecture and two of the behaviours of the task-level controller were adopted from a project conveyed last semester whereas the remaining behaviours and their prioritization scheme were designed during the current project. The framework was implemented in C++ and simulated with a model of a 7-DOF manipulator. The results of simulation demonstrated that the framework is capable to solve complex manipulation tasks in dynamic environments. The suggested framework is incomplete by design and several methods that could compensate for this drawback were proposed.

# 9. References

[1] D. Katz, J. Kenney and O. Brock, How Can Robots Succeed in Unstructured Environments?, *Workshop on Robot Manipulation: Intelligence in Human Environments at Robotics: Science and Systems, Zurich, Switzerland*, June 2008.

[2] O. Khatib, A unified approach for motion and force control of robot manipulators: The operational space formulation, *IEEE journal of robotics and automation*, vol. RA-3, no. 1, pp. 43-53, February 1987.

[3] L. Sentis and O. Khatib, Synthesis of whole-body behaviors through hierarchical control of behavioral primitives, *International journal of humanoid robotics*, vol. 2, no. 4, pp. 505-518, 2005.

[4] S. Pluzhnikov, Motion planning and control of nonholonomic mobile robot manipulators, *Specialization project report, NTNU, Department of Engineering Cybernetics*, 2011.

[5] Schunk GmbH. [Online]. http://www.schunk-modular-robotics.com

[6] B. Siciliano and O. Khatib, Eds., *Springer Handbook of Robotics*. Berlin: Springer, 2008.

[7] S. LaValle, Motion Planning , *IEEE Robotics & Automation Magazine*, vol. 18, no. 1, pp. 79 - 89, March 2011.

[8] L. Kavraki, P. Svestka, J. Latombe and M. Overmars, Probabilistic roadmaps for path planning in high-dimentional configuration spaces, *IEEE transactions on robotics and automatisation*, vol. 12, no. 4, pp. 566-580, August 1996.

[9] N. Amato., O. Bayazit, L. Dale, C. Jones and D. Vallejo, OBPRM: an obstacle-based PRM for 3D workspaces, *Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics: the algorithmic perspective*, pp. 155-168, 1998.

[10] C. Holleman and L. Kavraki, A framework for using the workspace medial axis in PRM planners, *IEEE Int. Conf. Robot. Autom.*, pp. 1018-1023, 1999.

[11] S. LaValle, M. Branicky and S. Lindermann, On the relationship between classical grid search and probabilistic roadmaps, *Proceedings of I. J. Robotic Res.*, pp. 673-692, 2004.

[12] R. Geraerts and M. Overmars, A Comparative Study of Probabilistic Roadmap Planners, *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR'02)*, pp. 43-57, 2002.

[13] Y. Yang and O. Brock, Efficient motion planning based on disassembly, *Proceedings of Robotics: Science and systems*, 2005.

[14] S. LaValle and J. Kuffner, "Rapidly-exploring random trees: progress and prospects", in *Algorithmic and Computational Robotics: New Direction*., 2001, pp. 293–308.

[15] K. Bekris, B.Y. Chen, A. Ladd, E. Plaku and L. Kavraki, Multiple query probabilistic roadmap planning using single query primitives, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 1, pp. 656 - 661, 2003.

[16] S. Karaman and E. Frazzoli, Incremental Sampling-based Algorithms for Optimal Motion Planning, *Robotics: Science and Systems (RSS)*, June 2010.

[17] S. LaValle. The RRT Page. [Online]. http://msl.cs.uiuc.edu/rrt/index.html

[18] B. Cohen, G. Subramania, S. Chitta and M. Likhachev, Planning for Manipulation with Adaptive Motion Primitives, *Proceedings of IEEE International Conference on Robotics and Automation ICRA '11.* , pp. 5478-5485, 2011.

[19] I. Sucan and L. Kavraki, Kinodynamic Motion Planning by Interior-Exterior Cell Exploration, *Algorithmic Foundation of Robotics VIII (Proceedings of Workshop on the Algorithmic Foundations of Robotics)*, vol. 57, pp. 449-464, 2009.

[20] E. Plaku and G. Hager, Sampling-based Motion and Symbolic Action Planning with Geometric and Differential Constraints, *IEEE International Conference on Robotics and Automation*, pp. 5002-5008, 2010.

[21] R. Rusu, I.A. Sucan, B. Gerkey, S. Chitta, M. Beetz and L. Kavraki, Real-time perception-guided motion planning for a personal robot, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4245 - 4252, 2009.

[22] O. Khatib, Real-Time Obstacle Avoidance for Manipulators and Mobile Robots, *IEEE International Conference on Robotics and Automation*, pp. 500-505, 1985.

[23] Y. Koren and J. Borenstein, Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation, *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1398-1404, April 1991.

[24] J. Borenstein and Y. Koren, Real-time Obstacle Avoidance for Fast Mobile Robots in Cluttered Environments, *IEEE International Conference on Robotics and Automation*, pp. 572-577, May 1990.

[25] S. LaValle, *Planning Algorithms*. Cambridge: Cambridge University Press, 2006.

[26] G. Fainekosa, A. Girard, H. Kress-Gazit and G. Pappas, Temporal logic motion planning for dynamic robots, *Automatica*, vol. 45, no. 2, pp. 343–352, February 2009.

[27] J. van der Berg and M. Overmars, Roadmap-Based Motion Planning in Dynamic Environments, *IEEE Transactions on Robotics*, vol. 21, no. 5, pp. 885-897, 2005.

[28] S. Karaman, M.R. Walter, A. Perez, E. Frazzoli and S. Teller, Anytime Motion Planning using the RRT*, *IEEE International Conference on Robotics and Automation*, pp. 1478 - 1483 , 2011.

[29] L. Jaillet and T. Siméon, A PRM-based Motion Planner for Dynamically Changing Environments, *International Conference on Intelligent Robots and Systems*, pp. 1606-1611, 2004.

[30] R. Gayle, K.R. Klingler and P.G. Xavier, Lazy Reconfiguration Forest (LRF) - An Approach for Motion Planning with Multiple Tasks in Dynamic Environments, *2007 IEEE International Conference on Robotics and Automation*, pp. 1316 - 1323, 2007.

[31] J. van den Berg, D. Ferguson and J. Kuffner, Anytime Path Planning and Replanning in Dynamic Environments, *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pp. 2366-2371, May 2006.

[32] D. Vasquez, F. Large, T. Fraichard, C Laugier and I. Rhone-Alpes, High-speed Autonomous Navigation with Motion Prediction for Unknown Moving Obstacles, *IEEE/RSJ international Conference on Intelligent Robots and Systems*, pp. 82-87, 2004.

[33] E. Owen and L. Montano, Motion planning in dynamic environments using the velocity space, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2833 - 2838 , 2005.

[34] Z. Shiller, F. Large, S. Sekhavat and C. Laugier, "Motion Planning in Dynamic Environments", in *Autonomous Navigation in Dynamic Environments*., 2007, pp. 107-119.

[35] N. Ratliff, M. Zucker, A. Bagnell and S. Srinivasa, CHOMP: Gradient optimization techniques for efficient motion planning, *Proceedings of IEEE International Conference on Robotics and Automation ICRA'09*, pp. 489 - 494, May 2009.

[36] C. Park, J. Pan and D. Manocha, ITOMP: Incremental Trajectory Optimization for Real-time Replanning in Dynamic Environments, *International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

[37] S. Quinlan and O. Khatib, Elastic bands: connecting path planning and control, *Proceedings to IEEE Int. Conf.Robot. Autom. (ICRA)*, vol. 2, pp. 802-807, 1993.

[38] N. Y. Ko, R. Simmons and D.J. Seo, Trajectory modification using elastic force for collision avoidance of a mobile manipulator, *Proceedings of the 9th Pacific Rim international conference on Artificial intelligence*, pp. 190-199, 2006.

[39] V. Delsart and T. Fraichard, Navigating Dynamic Environments Using Trajectory Deformation, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 226-233, 2008.

[40] R. Gayle, A. Sud, M.C. Lin and D. Manocha, Reactive Deformation Roadmaps: Motion Planning of Multiple Robots in Dynamic Environments, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3778-3783, 2007.

[41] O. Brock and O. Khatib, Elastic strips: a framework for motion generation in human environments, *International Journal of Robotics Research*, vol. 21, no. 12, pp. 1031-1052, December 2002.

[42] S. Petti and T. Fraichard, Safe Motion Planning in Dynamic Environments, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2210 - 2215 , 2005.

[43] J. Vannoy and J. Xiao, Real-Time Adaptive Motion Planning (RAMP) of Mobile Manipulators in Dynamic Environments With Unforeseen Changes, *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1199-1212, October 2008.

[44] Y. Yang and O. Brock, Elastic roadmap – motion generation for autonomous mobile manipulation, *Autonomous Robots*, vol. 28, no. 1, pp. 113-130, January 2010.

[45] P. Leven and S. Hutchinson, A Framework for Real-time Path Planning in Changing Environments, *The International Journal of Robotics Research*, vol. 21, no. 12, pp. 999-1030, December 2002.

[46] T. Kunz, U. Reiser, M. Stilman and A. Verl, Real-Time Path Planning for a Robot Arm in Changing Environments, *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2010, Taipei, Taiwan*, 2010.

[47] M. Kallmann and M. Mataric, Motion Planning Using Dynamic Roadmaps, *International Conference on Robotics & Automation*, pp. 4399-4404, April 2004.

[48] H. Liu, W. Wan and H. Zha, A Dynamic Subgoal Path Planner for Unpredictable Environments, *IEEE International Conference on Robotics and Automation*, pp. 994-1001, May 2010.

[49] K. Beevers and J. Peng, A* graph search within the BGL framework, *Boost Graph Library 1.33.0*, October 2003.

[50] M. Spong, S. Hutchinson and M. Vidyasagar, *Robot Dynamics and Control*, 2nd ed. New York: John Wiley and Sons, 2006.

[51] R. Featherstone, *Rigid Body Dynamics Algorithms*. New York: Springer, 2008.

[52] Bullet Dynamics Engine. [Online]. http://bulletphysics.com/

[53] Boost Graph Library (BGL). [Online]. http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/index.html

[54] Boost Serialization Library. [Online]. http://www.boost.org/doc/libs/1_49_0/libs/serialization/doc/index.html

[55] Eigen Libraries. [Online]. http://eigen.tuxfamily.org/

[56] Robot Operating System (ROS). [Online]. www.ros.org

[57] B. Bayle, J.-Y. Fourquet and M. Renaud, Manipulability of wheeled mobile manipulators: Application to motion generation, *International journal of robotics research*, vol. 22, no. 7-8, pp. 565-581, 2003.

# Appendix A. Contents of the digital attachment

The digital attachments enclosed contain the following materials:

- *Sources/* – the source files of the framework (C++)
- *Videos/* – video recordings of the experiments described in Chapter 5 (they are also available on YouTube)
- *Thesis.pdf* – a pdf-version of the current report.