Linköping University | Department of Computer and Information Science
Master's Thesis | Master of Engineering in Applied Physics and Electrical Engineering – International
Autumn term 2016 | LIU-IDA/LITH-EX-A--16/056--SE

Online Whole-Body Control using Hierarchical Quadratic Programming

Implementation and Evaluation of the HiQP Control Framework

Marcus A Johansson

Robert Krug, Örebro University
Cyrille Berger, Linköping University

LINKÖPINGS UNIVERSITET

Online Whole-Body Control using Hierarchical Quadratic Programming: Implementation and Evaluation of the HiQP Control Framework

at Centre for Applied Autonomous Sensor Systems
Örebro, Sweden

Marcus A Johansson

Master's Thesis in Applied Physics and Electrical Engineering
Linköping University, Sweden

November 7, 2016

Abstract

The application of local optimal control is a promising paradigm for manipulative robot motion generation. In practice this involves instantaneous formulations of convex optimization problems depending on the current joint configuration of the robot and the environment. To be effective, however, constraints have to be carefully constructed as this kind of motion generation approach has a trade-off of completeness. Local optimal solvers, which are greedy in a temporal sense, have proven to be significantly more effective computationally than classical grid-based or sampling-based planning approaches.

In this thesis we investigate how a local optimal control approach, namely the task function approach, can be implemented to grant high usability, extendibility and effectivity. This has resulted in the HiQP control framework, which is compatible with ROS, written in C++. The framework supports geometric primitives to aid in task customization by the user. It is also modular as to what communication system it is being used with, and to what optimization library it uses for finding optimal controls.

We have evaluated the software quality of the framework according to common quantitative methods found in the literature. We have also evaluated an approach to perform tasks using minimal jerk motion generation with promising results. The framework also provides simple translation and rotation tasks based on six rudimentary geometric primitives. Also, task definitions for specific joint position setting, and velocity limitations were implemented.

I'd like to thank

my supervisor Robert Krug at Örebro University
for his diligent explanations and helpful comments,

my examiner Cyrille Berger at Linköping University
for his guidance and support,

my family Lena and Urban, Martin and Cecilia
for their support and understanding.

Contents

Chapter 1

Introduction

1.1 Motivation

Apart from industrial robots who carry out very specific preset motions repeatedly, the next generation of service robots need to autonomously respond to changes and disturbances in their surrounding environment by adapting their movements. Examples of this involve humanoid walking on rough terrain in the DARPA Robotics Challenge [12], or Unmanned Aerial Vehicles responding to wind changes [8], or autonomous order picking in logistics [19]. Earlier approaches in artificial intelligence have aspired to discretize the configuration space to generate a graph and use graph-search algorithms like A-star to find a feasible path from a start node to a goal node [23]. However, such a classical artificial-intelligence-approach suffers from the curse of dimensionality[1] and renders a non-computable problem for most robot configurations. Another approach has been to use a sampling-based pose generator that randomly draws samples from the configuration space and tries to link such states together in some feasible way. Eventually as the network of configuration states grows a trajectory from a start pose to a goal pose is found and a desired motion can be executed. OMPL, for instance, is a framework that essentially generalises the way such sampling-based solvers are formalised [31]. The main problem with such a sampling-based approach is that it mostly results in suboptimal solutions as the finding of a feasible motion is very much depending on the random evolution of the state graph. Furthermore, the produced motion is seldom replicable and it is difficult to incorporate constraints in

[1] using breadth-first search algorithms to process the configuration space of for example humanoid robots often results in computational loads that are non-processable with today's computers

the motion generation problem formulation.

The idea that we explore further in this thesis is using motion generation and execution based on local optimal control. The controls are computed using a dynamic hierarchical optimization problem that is formed from finding redundancies in the desired end-effector pose placements. The redundancies are subject to the kinematic and/or dynamic constraints of the robot as well as customized constraints on desired evolution of the fulfillment of those end-effector positions. The redundancies are defined as functions of the configuration parameters in a so called *task space* and multiple redundancies are assembled to execute multiple motions simultaneously. Recomputing the controls is done in real-time and this approach allows for incorporation of sensor feedback which reduces any noise or environmental disturbances. Deviations from a desired motion are processed and handled in real-time. Our work has resulted in the HiQP kinematic control framework available with ROS support.

1.2 Problem Definition

The main objective of this thesis has been to investigate possible generalizations of motion primitives and motion generation by using local optimal control in a context of robotic grasping. Our work has resulted in the HiQP Control Framework, a whole-body motion generator built on the task function approach, see Section 3 for further details. The framework has been tested on the ABB YuMi robotic system.

The work was focused on the following research questions:

- How can motion primitives for robots be constructed from abstractions of pose and motion constraints ensuring minimal loss of kinematic redundancy?
- How can these abstractions be integrated in a scalable and dynamic way with emphasis on usability, intuitive design and fast performance?
- (optional) How can the task dynamics be formulated to allow for shared autonomy with human control interaction?
- (optional) How can the task dynamics be formulated to generate motions mimicking a learned expert behaviour, for example by using Dynamical Movement Primitives? [13]

Figure 1.1: Photo: ABB.

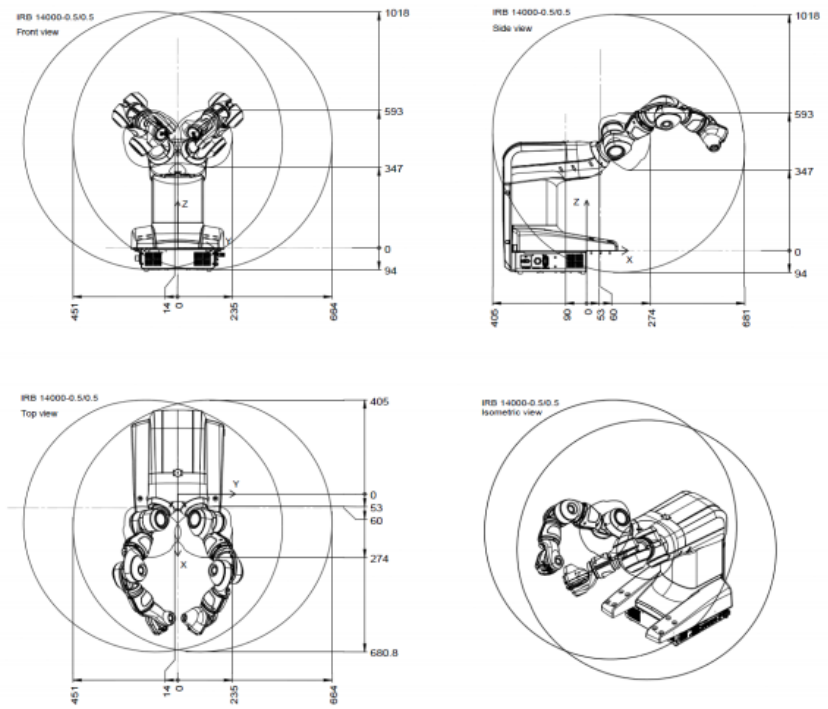
1.3 Thesis Outline

Chapter 1 introduces the area of study, describes the underlying dependencies of the framework, and defines the thesis project objectives. Chapter 2 accounts for related work in the area. Chapter 3 summarizes the main ideas in local control optimization that forms the theoretical basis for the framework. In Chapter 4 we present an overview of the software design of the framework. In Chapter 5 we describe our way of decomposing general motion primitives into geometric primitives and task definitions. Chapter 6 demonstrates two different areas of application of our framework. In Chapter 7 we evaluate the framework in terms of functionality and performance. In Chapter 8 we discuss the outcome of the evaluation data and relate it to our research questions. Chapter 9 concludes the thesis and suggests further work in the area.

1.4 The ABB YuMi Robot

The evaluation of the HiQP framework was performed on the YuMi robot from ABB [1], see Figure 1.1. The current version of YuMi only supports a joint position interface. All evaluations was done in simulation using a proprietary model of YuMi, Gazebo was used for physics simulation. To be able to set velocity controls for the robot a non-priority wrapper interface was used when running the framework on hardware. YuMi is a dual-arm manipulator robot with a gripper on the end-link of each arm. Each arm consists of seven joints.

Figure 1.2: Source: ABB.



Chapter 2

Related Work

Traditional AI approaches to motion planning have relied on the use of symbols to model the physical environment. Lozano-Pérez introduced configuration space representation in 1983 [22] which enabled classical AI search techniques once these *c-spaces* have been rasterized into bitmaps of collision/collision free intervals [21]. Such techniques involved dynamic programming, A*-search, best-first search etcetera to find a collision free path in the configuration space from an initial pose to a goal pose. Khatib (1986) [18] then introduced the use of potential fields in configuration space to generate robot controls in an attempt to battle high dimensionality in the search space. By placing artificial repulsive forces around obstacles and attractive forces around goals the control can be calculated from the gradient of the resulting potential field. One problem that arises from this method is that entering the vicinity of an obstacle could generate oscillations or make it impossible to enter into narrow passages [14]. Later in 1990-1991 Barraquand and Latombe [3] [4] presented an approach consisting of finding and connecting local minima of a potential function in the configuration space of the robot. The technique they used to escape local minima is based on Monte Carlo and Brownian motion execution. Zho and Latombe (1991) also addressed the popular *hierarchical approximate cell decomposition* method in [33]. This involves hierarchically decomposing the configuration space into rectangloid spaces known as cells. The cells are then marked as empty, full or mixed depending on the existence of obstacles inside the region. Hierarchical planning is performed in this manner by analysing a newly constructed cells connectivity to others in a connectivity graph. This graph is then used to search for obstacle free paths towards a goal. The primary problem though with classical grid search is that, as the number of dimensions of the c-space and the

resolution of the bitmap along each dimension increase, the search gets incalculable [21].

The graph-search based planner's suffering from a large number of decomposed cells, or from too many local minima in the potential field lead to mere practical use of these techniques on robots with up to five dimensions [15]. This inspired the development of sampling-based algorithms such as Probablistic Roadmaps (PRM) and Rapidly exploring Random Trees (RRT). Kavraki et.al. [16] submitted in 1996 the PRM technique for path planning. The original mechanism was divided into two steps: the learning and query phases. The objective of the learning phase is to build a collection of tree structures connecting collision free configurations with each other in feasible motion paths for the robot. This was done by randomly exploring the c-space adding new collision free configurations in the neighbourhood of old ones if a feasible path exists between them. The paths are generated by a local planner. The actual local path between nodes in the trees are not memorized, and no cycles are generated among the nodes. In the query phase, a start s and goal g configuration is connected by searching the trees. s and g are connected to nodes $\bar{s}, \bar{g}$ by the local planner. If a feasible path between $\bar{s}$ and $\bar{g}$ exists the path is returned, otherwise the query fails. As each query necessitates fast execution the strategy for connecting s and g to $\bar{s}$ and $\bar{g}$ is by considering nodes in increasing distance from s and g. Kuffner and LaValle [20] presented RRTs in 2000 which do not need knowledge about the environment a priori. Instead of making multiple queries searching in a world model that is generated offline, as for PRTs, RRTs generate two tree structures in the work space departing from the start and the goal configuration respectively. These trees are then randomly appended to with new configuration nodes using a heuristic. The work in [15] states that though sense-plan-act approaches (graph-search-based, or sampling based) have been shown to work well in practical implementations, and that they are both probablistically complete, they are limited by the dimensionality of the c-space of the robot. As the number of grid/sampling points grow exponentially with the number of dimensions, so does the worst-case running time of the planner.

2.1 Approaches Derived from Operational-Space Formulations

Berenson, Srinivasa and Kuffner (2011) have applied the task-space formulation on an RRT-based planning algorithm called CBiRRT2 and presented a framework [5]. In their work, they consider end-effector poses

as mappings from c-space to $SE(3)$, i.e. as positions and orientations in euclidean space. Much like our own work, they regard constraints on these poses and define Task Space Regions (TSRs) as the set of two transformation matrices in homogeneous coordinates from an initial frame to the end-effector pose and a 6×2-dimensional matrix of bounds on each DOF of the pose. Naturally, then these TSRs represent manifolds in C-space. Berenson et. al. then identify the applicability of the TSR formulation in sampling based planning algorithms by introducing a distance metric and sampling methods in task space. They note however that this can be problematic for kinematically redundant robots (number of DOF higher than 6) as the sampling inside a sub-manifold can bias the probablistic sampling in c-space. As the sampling with TSRs is done in task space, this can dramatically improve performance for highly redundant robots. To execute multiple tasks in parallel Berenson et.al. introduce TSR chains. A TSR chain is treated as a virtual kinematic structure defining dependencies between many TSRs and partakes in the motion planning as if they were real kinematic constraints. Berenson et.al. however identify a number of difficulties with this approach, namely: 1) difficult to incorporate non-holonomic constraints, 2) no current way of prioritizing constraints, 3) the sampling in task space biases the sampling in c-space. In contrast to our work a sampling based approach will be able to escape local minima, which the HiQP framework currently can not.

Chapter 3

Control-based Motion Generation and Execution

In this chapter we summarize the control and motion generation concepts that the HiQP framework is based on. The main idea is to formulate kinematic constraints in lower dimension submanifolds of the robot's configuration space. These constraints, together with the jacobians for these manifolds, are then used by an optimizer to compute velocity controls that will execute motions. Bu formulating these constraints in certain ways the resulting motions can be made to achieve certain robotic behaviours. We refer to these kinematic constraints as *tasks*, and *task dynamics*, and the rest of the chapter explains further these concepts.

3.1 The Task Function Approach

As the robot's configuration space is set under a number of kinematic constraints the actual space in which a motion is designed is a dimensionally smaller limited submanifold of the whole-body c-space [28]. Favourably then, one would like to operate in this smaller subspace when planning a motion; this idea forms the basis of the *task-function approach* [29], or the *operational space formulation* [17]. A task is defined as a triplet consisting of a *task function*, a *reference behaviour*, and a *differential mapping* between the whole-body space and a task space denoted $(e(q), \dot{e}(q)^*, Q(q))$ respectively.

Let us consider a robot with n joints, and a set of constraints limiting the configuration to a m-dimensional submanifold. Formally a *task function* is denoted as an m-dimensional vector function of the configuration

task function

vector q:

$$e(q) = 0 \tag{3.1}$$

where $q \in \mathbb{R}^n$, $e(q)$ is a function from $\mathbb{R}^n \mapsto \mathbb{R}^m$ [14].

Throughout this chapter we will return to the following example of a task given in [19]: Consider bringing an end-effector $p(q)$ onto a plane that is defined by the normal vector n and the offset distance d from the origin. The task function then becomes

$$e(q) = n^T p(q) - d \tag{3.2}$$

id est the projection residual between the plane and the end-effector minus the given distance d. In this manner the task is to bring the end-effector onto the plane, while the task function is a scalar function measuring the distance between the end-effector and the plane that we want to minimize. We will revisit this example throughout this chapter.

reference behaviour

The *reference behaviour* of a task denoted as $\dot{e}^*$ forms a relation on how the task behaviour $\dot{e}$ will develop over time. For example, when designing a movement for a robot arm, a nearby obstacle might be setup as a constraint on the task behaviour ensuring that the arm will not collide with the obstacle during execution of the task.

differential mapping

In order to fully define a task, one needs to specify the task space with respect to the configuration space. This is the *differential mapping*, Q, between the configuration space and the task space.

We can now depict the impact of a control u in the whole-body space by

$$\dot{e}(q) + \mu = Qu \tag{3.3}$$

where μ is known as the drift of the task, and Q is the *differential mapping* from the whole-body configuration space to the task space. The drift is not to be mistaken for sensor-drift or similarly in a traditional control sense, but is a consequence of a temporal change in the differentiable mapping introduced by higher-order derivatives. We will see later that for inverted kinematics (first-order derivation) μ becomes zero, but for inverted dynamics (second-order derivation) μ will not be equal to zero. Considering a desired reference behaviour $\dot{e}^*$ we get the appropriate control input u^* by taking the inverse of Q as follows.

$$\begin{aligned} u^* &= Q^{\#}(\dot{e}^* + \mu) + Pu_2, \\ \text{where } P &= I - Q^{\#}Q \end{aligned} \tag{3.4}$$

Here $Q^{\#}$ is any reflexive generalized inverse of Q, such as the Moore-Penrose pseudoinverse, and P is the projector onto the nullspace of Q

corresponding to $Q^{\#}$. u_2 therefore expresses the *redundancy* of the kinematic system of a robot corresponding to task e. **redundancy**

One of the main benefits of using the task function approach is that it enables a way to formalise a recursive use of the redundancies with respect to the degrees of freedom for various tasks. Particularly, redundancy expressed by the projection P can be used to successively process a strict hierarchy of tasks, or *Stack-of-Tasks* (SoT), by lending each lower priority task the remaining redundant degrees of freedom. This will lead to a recursive version of Equation (3.4) and we use k for ascending indexation of task functions $e_k(q)$ with descending priorities. Next, consider using the redundant control u_{k+1} to solve a new task $e_{k+1}(q)$ and solve the Equation (3.4) using the remaining control u_{k+1}. We get **Stack-of-Tasks**

$$\dot{e}_{k+1} + \mu_{k+1} - Q_{k+1}Q_k^{\#}(\dot{e}_k^* + \mu_k) = Q_{k+1}P_k u_{k+1}. \tag{3.5}$$

After deciding upon a reference behaviour for the propagation of task $k+1$, $\dot{e}_{k+1}^*$, we can derive the following *recursive control solution* **recursive control solution**

$$u_{k+1}^* = (Q_{k+1}P_k)^{\#}\left(\dot{e}_{k+1}^* + \mu_{k+1} - Q_{k+1}Q_k^{\#}(\dot{e}_k^* + \mu_k)\right) + P_{k+1}u_{k+2}. \tag{3.6}$$

[28] Later we will use this notation to form recursive SoT-solutions to the inverted kinematics and dynamics problem formulations.

Many tasks and task behaviour constraints are not defined as equalities with respect to the kinematic configuration. Some examples include observing joint limits, avoiding obstacles, or the goal state being a set of multiple feasible configurations et cetera. In [14] the writers therefore extend the task definition from Equation (3.1) to include *inequality tasks* as **inequality tasks**

$$e(q) \leq 0. \tag{3.7}$$

This reduced form also encompasses lower bounds $e(q) \geq 0$, double bounds $-r \leq e(q) \leq r$, and equalities $0 \leq e(q) \leq 0$. Regarding *inequality constraints* we use the same reduced form: **inequality constraints**

$$\frac{\partial e}{\partial q}\dot{q} \leq 0 \tag{3.8}$$

as this also includes lower and double bounds as well as equalities. The Stack-of-Tasks solution to this problem tries to fulfill lower-level tasks as good as possible in the Least Squares sense inside the null-space of higher-level tasks [24]. In general, problem definitions with inequalities cannot be solved using the inverse of Q. One approach proposed by [14] instead adds *slack variables* to the inequalities reorganizing them into equalities and then relies on quadratic programming solvers, see Section 3.4, for a resolution. **slack variables**

3.2 Inverted Kinematics

The concept of the task function approach can be used to solve general inverted kinematics problems by explicitly formulating the task behaviour and the differential mapping between the configuration space and the task space. In inverted kinematics mode, the control is intuitively chosen as the velocities of each joint $u = \dot{q}$ [28]. The *task evolution*, or equivalently the task behaviour [28], is then derived by differentiation

task evolution

$$\dot{e}(q) = J\dot{q} \tag{3.9}$$

$$J = \frac{\partial e}{\partial q} \tag{3.10}$$

where J is the *task jacobian* [19] and expresses the differential mapping between the two spaces, id est J corresponds to Q in the template in Equation (3.3) $Q = J$ [28]. The main idea behind inverse kinematics is to find joint velocities which produce given task space velocities. The term *inverted* therefore originates from

task jacobian

$$\dot{q} = J^{-1}\dot{e}^* \tag{3.11}$$

which says that the joint velocities can be obtained from the differentiable mapping J and a reference behaviour $\dot{e}^*$ described in a task space.

By matching Equation (3.9) with the template in Equation (3.3) we get that

$$Q = J, \;\; u = \dot{q}, \;\; \mu = 0. \tag{3.12}$$

This can now be inserted in template Equation (3.6) yielding

$$\begin{gathered} \dot{q}^*_{k+1} = (J_{k+1}P_k)^{\#}(\dot{e}^*_{k+1} - J_{k+1}J_k^{\#}\dot{e}^*_k) + P_{k+1}\dot{q}_{k+2} \\ P_k = I - Q_k^{\#}Q_k. \end{gathered} \tag{3.13}$$

Choosing a Reference Behaviour

The differential mapping Q is essentially a consequence of how the task function $e(q)$ is defined. In the example of an end-effector begin at a given distance to a plane, see Equation (3.2), by differentiating the task function we get the following relation.

$$\begin{aligned} e(q) &= n^T p(q) - d \\ \dot{e}(q) &= n^T \frac{\partial p}{\partial q}\dot{q}. \end{aligned} \tag{3.14}$$

It is clear that $Q = n^T \frac{\partial p}{\partial q}$, in other words that the task jacobian in this case is time dependent through the configuration q. By inverting this kinematic relation we get

$$\dot{q} = (n^T \frac{\partial p}{\partial q})^{-1} \dot{e}(q) \tag{3.15}$$

The reference behaviour defines the desired task dynamics, in other words how the qualities describing the task are supposed to change over time. In this section we will discuss one common way of defining a reference behaviour in particular and comment on other possible behaviours. However one must keep in mind that stability of the reference behaviour is crucial to the outcome of a task-function based controller.

A common practice is to let the reference behaviour $\dot{e}^*$ cause an exponential decay of e to zero [28], forming the *Ordinary Differential Equation* (ODE)

Ordinary Differential Equation

$$\frac{\partial e_k}{\partial q} \dot{q} = -\lambda e_k(q) \tag{3.16}$$

where λ is a positive real constant. The decay to zero eventually leads to fulfilling the task constraint $e_k(q) = 0$, which motivates this approach [14]. This setup also facilitates the handling of inequality constraints as we will discuss forthwith.

By relying on the system interpretation in Equation (3.16) we derive the *Ordinary Differential Inequality*

Ordinary Differential Inequality

$$\frac{\partial e_k}{\partial q} \dot{q} \leq -\lambda e_k q \tag{3.17}$$

Such an inequality lends itself well to solving by using *Grönwall's Inequality* which is presented in [14].

Lemma 1 (Grönwall's Inequality on Differential Form)**.** *Let $f \in C^1([a, b])$ and $g \in C([a, b])$, where a and b are real constants, such that*

$$f'(t) \leq g(t) f(t) \quad \forall t \, : \, a < t < b$$

then

$$f(t) \leq f(a) e^{\int_a^t g(\tau) d\tau} \quad \forall t \, : \, a < t < b$$

□

Here C is the set of continuous functions, and C^1 is the set of continuous functions with continuous first derivatives with respect to time. The ordinary differential inequality in Equation (3.17) can then be solved and we write

$$e_k(q) \leq e_k(q_0) e^{-\lambda(t - t_0)} \tag{3.18}$$

where $t > t_0$ and $q_0 = q(t_0)$. We examine again the example task given earlier on keeping an end-effector $p(q)$ at a distance d from a plane with normal vector n, making the task function $e(q)$ being the distance between the end-effector and the plane. According to the first order dynamics reference behaviour chosen in Equation (3.16), and under the assumption that the initial distance is 1.8 length units, the desired distance $d = 0$, $t_0 = 0$ and $\lambda = 2$, the task function is then limited by the inequality $e(q) \leq 1.8e^{-2t}$. If we plot this equation we can see that the reference behaviour results in an infinite acceleration spike at the beginning and a slow motion towards the goal state at the end, see Figure 3.1. We can conclude that

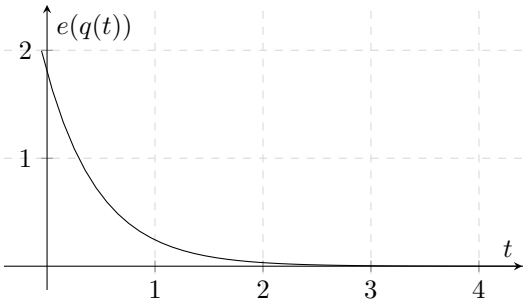


Figure 3.1: The distance between the end-effector $p(q)$ and the plane over time considering the reference behaviour given in Equation (3.18).

both equality and inequality constraints converge exponentially fulfilling the task formulated by the set of constraints [14].

minimal-jerk hypothesis

According to the *minimal-jerk hypothesis* in [32], humans try to perform a movement by minimizing the square of the jerk[1] over the whole movement. One interpretation of such an objective is that it will result in a minimization on the changes of stresses on the parts of a moving body. Such a criterion is derived by [9] resulting in

$$q_i(t) = q_i^{(0)} + (q_i^{(0)} - q_i^{(f)})(15\tau^4 - 6\tau^5 - 10\tau^3) \tag{3.19}$$

where $\tau = \frac{t}{t_f}$, $q_i^{(0)}$ is the i:th joint's position of the robot at the initial time $t = t_0$, and $q_i^{(f)}$ is the i:th joint's position of the robot at the final time $t = t_f$.

[1] Jerk is the rate of change of acceleration with respect to time.

3.3 Inverted Dynamics

Following the notation used in inverse kinematics we can broaden our task domain to incorporate also the dynamics of a robot. We find the second derivative of the task function from the inverted kinematics notation, see Equation (3.9), by writing

$$\ddot{e} = J\ddot{q} + \dot{J}\dot{q}. \tag{3.20}$$

Considering the robot in free space the joint accelerations $\ddot{q}$ can then be derived from for example Newton's and Euler's laws of motion:

$$M(q)\ddot{q} = \tau + f(q, \dot{q}) \tag{3.21}$$

where $M(q)$ is the inertia matrix of the dynamic system, τ are the torques acting on each joint, and $f(q, \dot{q})$ is the sum of the nonlinear Coriolis, centrifugal and gravitational forces on the system. By plugging Equation (3.21) into Equation (3.20) we get:

$$\ddot{e} - \dot{J}\dot{q} + JM^{-1}f(q, \dot{q}) = JM^{-1}\tau. \tag{3.22}$$

In analogy with setting $u = \dot{q}$ in inverted kinematics to get the joint velocities from a given reference behaviour $\dot{e}^*$ we can interpret $u = \tau$ as our controls in inverted dynamics. By fitting this interpretation with a second-order differentiation of the template from Equation (3.3) we get:

$$\mu = -\dot{J}\dot{q} + JM^{-1}f(q, \dot{q}) \tag{3.23}$$

$$Q = JM^{-1} \tag{3.24}$$

This can then be inserted in Equation (3.6) to get a recursive solution to τ^*_{k+1} from a reference behaviour $\ddot{e}^*_{k+1}$ with respect to the remaining redundancies of the robot manipulators.

3.4 Quadratic Programming

As stated above, the solution by inverting the differential mapping of each task is only possible for tasks and task constraints on equality form. One instead leans on quadratic programming solvers to relax different constraints of a task to get a Least Squares solution to the problem. In this section we will consider the QP-solution to a problem with only equality constraints, and extend to a method used in [14] to solve for inequality constraints. Later we shall also summarize a hierarchical quadratic

problem resolution to solve a Stack-of-Tasks (SoT) with strict descending priorities using the remaining redundant degrees of freedom.

We start by recalling the definition of a *quadratic program* and the derivation of its solution. A quadratic program is an optimization problem with a quadratic objective function and linear constraints. A general quadratic program with constraints of *equality form* can be written in matrix notation as

quadratic program

equality form

$$\begin{aligned} \min_{x} \quad & q(x) = x^T c + \frac{1}{2} x^T G x \\ \text{s.t.} \quad & Ax = b \end{aligned} \tag{3.25}$$

The *first-order optimality conditions*, the Karush-Kuhn-Tucker conditions, state that the gradient of the objective function must be a linear combination of the gradients of each constraint given in $Ax = b$ together with x^* being a feasible solution with respect to the constraints. This is written

first-order optimality conditions

$$\nabla_x q(x)|_{x=x^*} = \sum_{i=1}^{m} \lambda_i \nabla g_i(x) \tag{3.26}$$

where $g_i(x)$ is the *i:th* equality constraint in (3.25), and λ_i are the *Lagrangian multipliers*. These conditions can be written as a system of linear equations:

$$\begin{bmatrix} G & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix}. \tag{3.27}$$

This is known as a *Karush-Kuhn-Tucker system*. We can see that if A has full rank there exists a unique solution (x^*, λ^*) to Equation (3.27). Further, if the *second-order optimality conditions* are fulfilled, that is the second gradient of the objective function at x^* being larger than zero, then (x^*, λ^*) is a globally minimal solution to the quadratic program in Equation (3.25). The second-order optimality conditions are written as

Karush-Kuhn-Tucker system

$$Z^T G Z > 0 \tag{3.28}$$

where Z is a matrix whose columns form the nullspace of G such that Z has full rank and $GZ = 0$. In other words, if G is *positive semidefinite* ($x^T G x > 0 \ \forall x \neq 0$) the solution (x^*, λ^*) will be a globally minimal solution. G being positive definite makes the quadratic program in Equation (3.25) convex. [26] [6]

positive semidefinite

Solving for Inequality Constraints

The problem formulation in Equation (3.25) can be extended to allow for having inequality constraints as follows

$$\begin{aligned} \min_{x} \quad & q(x) = x^T c + \frac{1}{2} x^T G x \\ \text{s.t.} \quad & Ax = a \\ & Bx \geq b. \end{aligned} \tag{3.29}$$

Considering a point x^*, the *active set* of x^* is denoted **active set**

$$\mathcal{A}(x^*) = \{i : B_i^T x^* = b_i\} + \mathcal{E} \tag{3.30}$$

where B_i and b_i are the rows of B and b in Equation (3.29), and $\mathcal{E}$ are the set of indices of all equality constraints. In other words the set of indices of all the inequality constraints equaling to zero at x^*. Solving the general optimization problem in Equation (3.29) now corresponds to finding the *optimal active set* and solve it as a problem with only equality constraints. We will now explore some of the most common approaches to finding such an optimal active set.

Active Set Methods

1. Start with choosing a subset of all inequality constraints known as the *working set*.
2. Try to guess a solution x_k and let $p = x - x_k$ where x is the actual solution to the main problem.
3. Reformulate the optimization problem as a problem to solve p instead of x over only the *working set* regarded as active equality constraints.
4. If $p = 0$ and if the Lagrangian multipliers associated with the inequalities on the current working set are all greater than or equal to 0 we have found the optimal active set.
5. If $p = 0$ and some Lagrangian multiplier is less than zero we remove the inequality associated with the lowest multiplier and continue from 3.
6. If $p \neq 0$ we compute a step length α and we set $x_{k+1} = x_k + \alpha p$. If this step involves violating a constraint we add that constraint to the working set. Continue from 3.

Such a solver can be implemented as a linear-equation-system solver [26]. Equation (3.27) can be adjusted with the variable substitution $x^* = x - p$ where x is a solution estimate and p being the desired step. Such a formalism enables computing the solution through iteration. The linear equation system can now be obtained as

$$\begin{bmatrix} G & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} -p \\ \lambda^* \end{bmatrix} = \begin{bmatrix} c + Gx \\ Ax - b \end{bmatrix}. \tag{3.31}$$

3.5 Solving a Stack-of-Tasks Problem

The task-function-approach relies on solving lower priority tasks in the null-space of higher priority task solutions. In this section we summarize the concepts used in solving such a hierarchical optimization problem.

A *stage* is the set of tasks with the same priority level among all currently active tasks. At each time step of the controller all tasks are assembled in stages forming stacks of vectors and matrices resulting in one compound task for each stage. A stage only contains the jacobians and task dynamics matrices/vectors since the task function values are not used in solving the final optimization problem. Equation 3.32 depicts an example of a stage containing two tasks.

$$\begin{aligned} \boldsymbol{J}_{stage} &= \begin{bmatrix} \boldsymbol{J}_1 & \boldsymbol{J}_2 \end{bmatrix}^T \\ \dot{\boldsymbol{e}}^*_{stage} &= \begin{bmatrix} \dot{\boldsymbol{e}}^*_1 & \dot{\boldsymbol{e}}^*_2 \end{bmatrix}^T \end{aligned} \tag{3.32}$$

When the stages are assembled for each iteration the tasks are reformulated as less-than inequality tasks, i.e. $\dot{\boldsymbol{e}}^* \leq 0$, by possibly altering the sign of the task function. Equality tasks are added twice with opposite signs. In order to solve for optimal controls (velocity, but could also be done for effort) the stages are formed into quadratic problems by setting the following constraints

$$\boldsymbol{J}_k \dot{\boldsymbol{q}} \geq \dot{\boldsymbol{e}}^*_k + \boldsymbol{w}_k \tag{3.33}$$

where k is the index of each stage and is directly related to the priority level, and $\boldsymbol{w}_k$ are slack variables of that stage. This relation is then used in a recursive hierarchical structure where the slack variable is fixed between stages. We write the final recursive optimization problem as:

$$\begin{aligned} \min_{\dot{\boldsymbol{q}}, \boldsymbol{w}_p} \quad & \frac{1}{2}||\boldsymbol{w}_p||_2^2 + \kappa||\dot{\boldsymbol{q}}||_2^2 \\ \text{subject to} \quad & 0 \leq \boldsymbol{J}_i \dot{\boldsymbol{q}} - \dot{\boldsymbol{e}}^*_i - \boldsymbol{w}_i \leq \infty \\ & 0 \leq \boldsymbol{J}_p \dot{\boldsymbol{q}} - \dot{\boldsymbol{e}}^*_p - \boldsymbol{w}_p \leq \infty \\ & \text{where } \; i = 1, ..., p-1 \end{aligned} \tag{3.34}$$

This is the same as trying to find both the control output, $\dot{\boldsymbol{q}}$ in the kinematic case and $\boldsymbol{\tau}$ in the dynamic case, and the slack variable $\boldsymbol{w}_p$ that minimizes the L_2-norm of the slack variables of each stage $p = 1, 2, ..., P$. The control vector $\dot{\boldsymbol{q}}$ or $\boldsymbol{\tau}$ retrieved at the lowest priority level then is the instantaneous optimal solution of the given task hierarchy for a particular robot configuration. The slack variables of upper and already solved stages are locked when processing lower priority stages, this ensures that the solver operates in the null-space of higher tasks.

Chapter 4

System Architecture: Quality and Design

4.1 Design Principles

Upon designing the framework we relied on three areas of software design: a set of general software quality attributes [2] [30], four common requirements modelling techniques [27] [30], and the five *SOLID* class design principles [25]. We have chosen only a subset of the common practices in software engineering that we saw most fit to our needs. This section traces the three areas and explains our view on their application.

As we wanted to rely on a set of quality objectives when designing and implementing the framework, a short survey on the area of software quality was conducted. According to [7] in general various *-ilities* (functionality, reliability, usability etc.) outline the overall quality attributes and these often variy between quality studies and software designs. However, the particular terminology chosen often targets the same comparable aspect of a software quality attribute.

A comprehensive set of quality attributes and informal definitions is given in [2]: *reusability*, *flexibility*, *understandability*, *functionality*, *extensibility*, *effectiveness*. A slightly modified version of these attributes are what we have leaned on in the design process of our framework, see Table **??**. However, we have not cared for any of the software security issues discussed in [2].

In addressing these attributes in the design of the control framework we looked at a set of requirements modelling methods, see Table **??**. As with the quality attributes, many different modelling methods exist in the

QUALITY ATTRIBUTES	
Reusability	Relates to how well a design structure allows for reapplication to new problems without significant change. Such a portable design is easy to change or adapt upon replication.
Flexibility	Relates to how resistant in degradation the design is to structural changes. A flexible design is able to adapt to suit a similar functionality without much effort. A high level of coupling among distinct modules can lead to low flexibility.
Understandability	Relates to which degree the design is comprehensible in terms of its complexity. This incorporates how easy the framework is to install, use and extend.
Functionality	Relates to how well the responsibilities of classes suit the problems they intend to solve. Such an attribute encourages modularity in the design to allow for interoperability, and also pursues testability.
Extensibility	Relates to how open the design is to extensions in terms of new requirements.
Effectiveness	Relates to what degree the design is able to achieve the desired functionality, particularly in terms of computation time and memory usage. An effective design is also regarded as to a degree self-recoverable in being fault-tolerant.

Table 4.1: The set of quality attributes intended for the design of our control framework.

literature [30] that targets the same areas. We refer to one definition of a chosen subset of these modelling methods in Table **??**, and we discuss the outcome of these models in Section 4.2.

Upon identifying the general requirements on the framework we made use of the class-design principles in [25] and the design pattern found in [11]. The SOLID class-design principles in [25] are cited in Table 4.2 for easy reference. As our plan was to write one single package for our control framework, we never concerned ourselves with any of the package cohesion and coupling principles that were also presented in [25]. The actual design patterns that were applied, along with the interpretation of the class-design principles in practice, is presented with the framework structure in Section 4.3.

The Single Responsibility Principle
— A class should have one, and only one, reason to change.

The Open Closed Principle
— You should be able to extend a classes behavior, without modifying it.

The Liskov Substitution Principle
— Derived classes must be substitutable for their base classes.

The Interface Segregation Principle
— Make fine grained interfaces that are client specific.

The Dependency Inversion Principle
— Depend on abstractions, not on concretions.

Table 4.2: The S.O.L.I.D. software design principles related to class design as stated in [25].

4.2 Requirements Elicitation

We start this section by informally describing the setting in which the framework is supposed to live. This is a result of trying to apply the system modelling methods described in Table **??**.

The main user-groups this control framework is intended for are scientific- and industry-affiliated teams developing robotic and/or autonomous control systems. The framework is not thought of as a highly scalable and fully industrial tool, but rather an open source contribution preferably

REQUIREMENTS MODELLING	
Enterprise Modelling	Used to describe the organizational objectives to investigate how the software should be operated. A functional approach focusing on the role of the software and its purpose.
Data Modelling	Considers what data the software needs to represent and how it is to be presented to the user(s). Values high comparability between the data handled and the real world phenomena it represents.
Behavioural Modelling	Models how the system logically behaves in terms of data handling, and modular interaction. A type of modelling that outlines the software's behavioural functionality.
Domain Modelling	Attempting to detail the technical domain in which the software will live in terms of domain assumptions.

Table 4.3: The requirements elicitation models, [27], that we based our list of requirements on.

supporting or highly integrated with the ROS framework. Direct users of our framework are experienced C++ developers. The control framework shall aim to allow users to easily and efficiently define own tasks for any robot configuration and to provide the user with an efficient inverted kinematics/dynamics solver. The tasks shall be easy to add, change or remove on demand at run-time and defining uni- and double-bounded inequality constraints and equality constraints shall be made straightforward to the user. Different optimization back-ends such as Gurobi, IPOPT, or proprietary libraries shall be easily swapped, making possible investigation on different solvers behaviour; this is not required to be available at run-time. Defining own tasks or changing which optimizer back-end to use could require C++ knowledge, setting tasks at run-time shall not.

Dealing with robot manoeuvring, the main data that needs to be represented in the framework are the robot configuration and the task function, reference behaviour, differential mapping and task drift at each time instant. Regarding the configuration, this would preferably be represented using some data structure that is already commonly used in the ROS community. The task-related metrics are preferably represented as scalar matrices as they simply are snapshots of the values at each sampling time step in the controller and evolves due to the definition inside a user-defined task implementation. The framework thus needs to be given the current robot configuration at each time step, and calculates the current task metrics which are then given to a optimizer that generates the joint velocities for a kinematic controller, or joint torques for a dynamic controller.

As task-setting shall be accessible at run-time, this functionality shall be offered through some facility well-known in the ROS community. Setting a task shall allow specifying its intrinsic parameters which requires a rather generic way to do this. Also the user shall be able to set the priority of any task, change the priority, start and pause the task and remove tasks at run-time. Upon faulty input, each task is alone responsible for the handling of it. However, the control framework shall not be allowed to cause the whole controller to crash upon a user-defined task not being able to handle erroneous information. Erroneous information shall instead be printed to the standard output stream as a warning message.

4.3 Framework Architecture

Coming out of the requirements identified in Section 4.2 we suggested a set of features. These features have been made accessible via the HiQP framework and are listed below.

1. Add new tasks by inheriting from one or more common task inter-

faces. This is to allow the user to formulate own task spaces, task functions and task dynamics.

2. Add and remove tasks during run-time through ROS service calls.
3. Use geometric primitives when working with existing tasks or when defining new ones. This is to facilitate describing tasks as constraints related to geometric objects.
4. Add and remove geometric primitives during run-time through ROS service calls.
5. Use a basic set of robot movement tasks without knowledge of the inner workings of the framework.
6. Be able to change the solver used through CasADi, or implement his/her own solver.
7. Visualize tasks and geometric primitives in rViz.

We will continue with a discussion on the framework design in the context of each of these features and relate the decisions made to the quality attributes and the class-design principles stated in Section 4.1.

4.3.1 Architecture Overview

The framework is built around the *Model-View-Controller* design pattern where the Task Manager acts as the *Controller*, or as a *mediator*, connecting the other classes; please refer to Figure **??**. The `HiQPKinematicsController` and the `HiQPDynamicsController` are derived from ROS specific interfaces wrapping the dependency on ROS. ROS then communicates with the framework through these classes. Along with the `Visualizer` interface and the `ROSVisualizer` realization, these classes forms the *View* of the MVC pattern in that the data produced by the framework, the actual joint controls and visualization messages, are delivered by these classes. The `TaskFunction`, `TaskDynamics` and `HiQPSolver` interfaces along with their realizations form the *View* part of the MVC pattern. The realizations of these interfaces do however perform computations and data processing such as computing task function and jacobian values, and solve quadratic programs.

In compliance with the fifth SOLID principle, all classes except for the Task Manager are realizations of interfaces with a clear functional contract. This increases the reusability of the code, both among the *View* classes, and among the *Model* ones, by enabling further realizations. The specific ROS realizations, i.e. the two controller classes, can be swapped

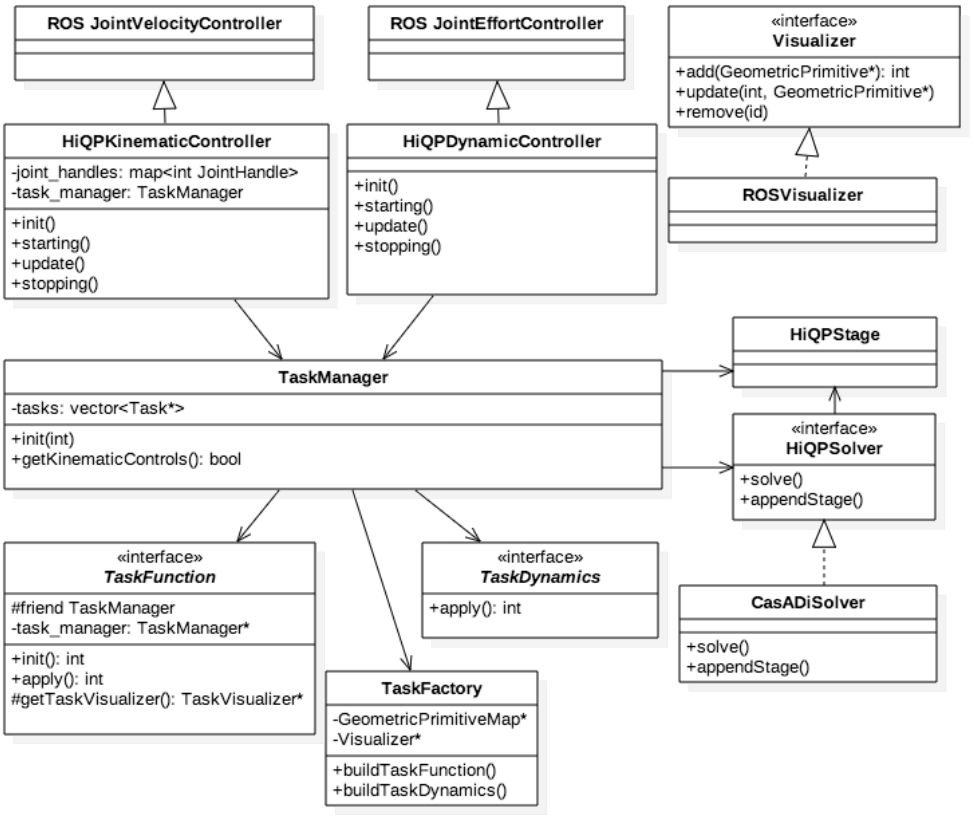


Figure 4.1: UML diagram showing an overview of a selected part of the system architecture. Not all classes are visible here, and not all member methods and inter-class dependencies are shown.

with classes that realizes some other interface according to another communication bridge than ROS. The same holds true for the `ROSVisualizer` class. In this manner, the Task Manager can be seen as the entry point of the framework in that only by communicating with the Task Manager a program can gain access to the full capability of the framework.

In an attempt to reduce class coupling to increase flexibility, the Task Manager provides three ways of communication: one towards the ROS wrapper classes, one towards the quadratic programming solver implementations, and one towards the task defining classes. These three divisions of data-flow in the framework clarify the division of functionality which makes the framework more understandable to the user.

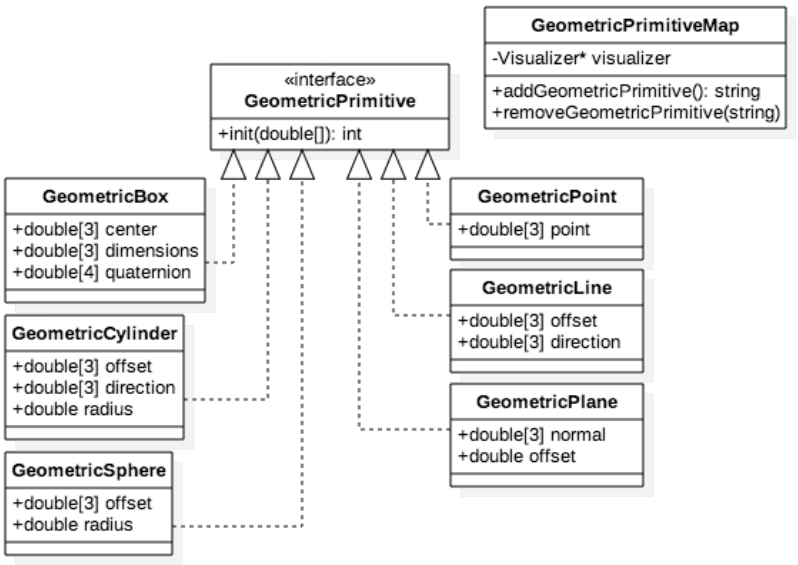


Figure 4.2: Geometric Primitives Class Structure.

The framework also provides a set of six geometric primitives: a point, a line, a plane, a box, a cylinder and a sphere primitive. The choice of this set of primitives is to provide zero-, one-, two- and three-dimensional geometric representations. These types are defined as realizations of the `TaskGeometricPrimitive` interface and can be either accessed by the user through the `GeometricPrimitiveMap`, or by using the type constructors directly. By using the Geometric Primitive Map instance when creating

geometric primitives, the primitive is mapped to a string identifier and available in all parts throughout the framework. The primitives themselves are agnostic to any visualizer and therefore unable to visualize themselves. This functionality is provided by the Geometric Primitive Map class and primitives created through its interface can be sent to the visualizer in use. The Geometric Primitive Map is also made accessible through ROS service calls and primitives can be created at run-time. The user can then implement his/her own task and use the string identifier of a primitive to access it via the map.

To handle the kinematic structure of an arbitrary robot inside the framework we rely on *KDL*[1] from Orocos. KDL provides functionality to parse a ROS robot description and uses its own indexing of the joints called `q_nr`. Since we have made the interface towards the communications system, in this version of the implementation ROS, modular with respect to the `TaskManager` class we want to keep the joint indexation in ROS separate from the `q_nr` used by KDL. This is the motivation behind adding the `joint_handles_map_` field in `RosKinematicsController`.

4.3.2 Custom Task Implementation

The `TaskFunction` interface has three pure virtual methods *init*, *apply* and *monitor*, and one virtual method *getFinalState*, see Figure **??**. The task function value $\boldsymbol{e}$ is stored as a protected member of type `Eigen::VectorXd` in `TaskFunction`. The task jacobian $\boldsymbol{J}$, the desired task dynamics $\dot{\boldsymbol{e}}^*$, and the initial and final task function values are also stored in this way. The init-method is supposed to be implemented as setting the correct sizes of these member fields and to set the initial values for them. The apply-method is called at every time step of the controller and should update the values of these data fields which are later collected to form the stages that are sent to the solver. The monitor-method can be used to produce any specific performance measures that are to be sent along with the monitoring topic published by the `ROSKinematicsController` class. The getFinalState-method returns the zero-vector $\mathbf{0}$ as a `Eigen::VectorXd` with the same size as $\boldsymbol{e}$ by default but can be reimplemented to allow for other final task function values than zero. This is mainly used by task dynamics that are non-holonomic. The same methods for `TaskDynamics` have the same intent as given above, see Figure **??**

4.3.3 Interaction at Run-Time

Add a geometric primitive:

[1] Kinematics and Dynamics Library

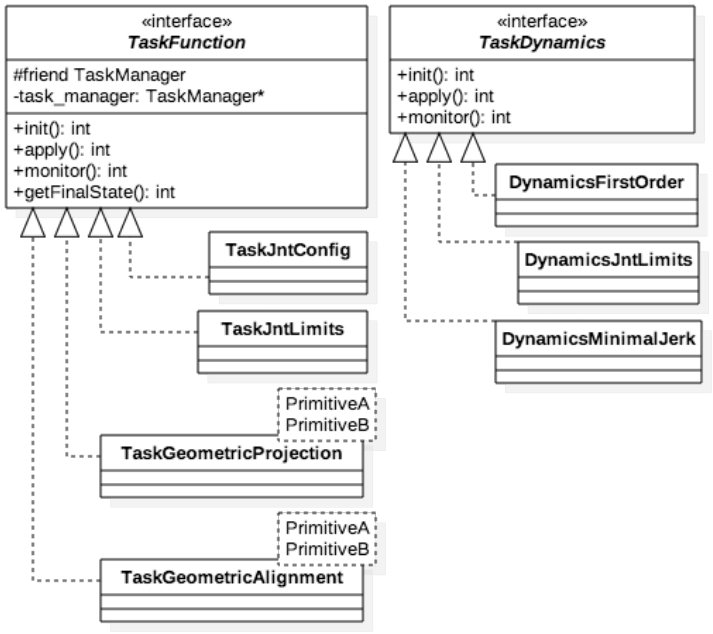


Figure 4.3: Implementations of the `TaskFunction` and `TaskDynamics` interfaces.

```
rosservice call /yumi/hiqp_kinematics_controller/add_primitive
    \
"name: 'mypoint1'
type: 'point'
frame_id: 'gripper_r_base'
visible: true
color: [1.0, 0.0, 0.0, 0.9]
parameters: [0, 0, 0.1]"
```

Add a task:

```
rosservice call /yumi/hiqp_kinematics_controller/add_task \
"name: 'geomproj155'
type: 'TaskGeometricProjection'
behaviour: ['DynamicsFirstOrder', '0.5']
priority: 1
visibility: 0
```

```
parameters: ['point', 'box', 'mypoint1 = mybox1']"
```

Remove a certain geometric primitive:

```
rosservice call /yumi/hiqp_kinematics_controller/
    remove_primitive 'name: mypoint1'
```

Remove a certain task:

```
rosservice call /yumi/hiqp_kinematics_controller/remove_task '
    task_id: 2'
```

Remove all primitives:

```
rosservice call /yumi/hiqp_kinematics_controller/
    remove_all_primitives
```

Remove all tasks:

```
rosservice call /yumi/hiqp_kinematics_controller/
    remove_all_tasks
```

4.3.4 CasADi and Solving Optimal Controls

The `TaskManager` searches at each time step for all active tasks and orders them with respect to their priority level. The manager then creates new `HiQPStage` objects, one for each priority level apparent among the active tasks, and copies the data from the task objects to the stages. The assembly of the stages are done in likewise to the way described in Section 3.5.

Chapter 5

Task Specific Implementations

In this chapter we lay out the details on specific task implementations that follow with the HiQP framework. We note that when implementing a task, the task function and task jacobian expressions must be expressions of the same behaviour. However, as in some of our own cases in this chapter, having a proportionality constant in the jacobian is valid and only affects the speed of the task behaviour.

5.1 Joint Configuration

A joint configuration task is useful when debugging a controller, it is easier to bring the robot back to an initial state between running tasks than to restart the controller. Setting the total robot configuration could be a useful feat in-between stages of a finite-state-machine to ensure a certain pose before execution a next set of tasks.

We identify two ways of implementing this functionality as a task:

1. by designing the joint configuration task function as simply the sum of squares of the deviations from the desired joint position yielding a one-dimensional task function,

2. or, as the vector difference between a desired joint position vector and the current joint position vector, yielding a n-dimensional vector where n is the number of joints.

The second design choice will not require a recomputation of the task jacobian as it always becomes the n-by-n identity matrix, while the former

requires such recomputation. The task progression does not depend on which of the designs is used and therefore the design choice boils down to a matter of computational efficiency. Recalculating the 1-by-n jacobian vector takes $\mathcal{O}(n)$ time in the former design alternative, while extending the task dimensionality to use n slack variables in the quadratic programming solver always takes at least $\mathcal{O}(n)$ to compute. We have therefore chosen the first design alternative.

The task function the becomes

$$e = (\boldsymbol{q}^* - \boldsymbol{q})^T(\boldsymbol{q}^* - \boldsymbol{q}) \tag{5.1}$$

where $\boldsymbol{q}^*, \boldsymbol{q} \in \mathbb{R}^{n\times 1}$. The time derivative of this task function becomes

$$\dot{e} = -2(\boldsymbol{q}^* - \boldsymbol{q})^T\dot{\boldsymbol{q}} \tag{5.2}$$

and we write the task jacobian as

$$J_e = -2(\boldsymbol{q}^* - \boldsymbol{q})^T \tag{5.3}$$

where $J_e \in \mathbb{R}^{1\times n}$.

When this task formulation is used as an inequality task $\boldsymbol{q}^*$ functions as a lower or upper bound for joint positions and therefore suits well to the design of obstacle avoidance tasks.

A sample service call to start a joint configuration task, for a 5-joint robot, is given below.

Figure 5.1: A sample ros service call to start a joint configuration task.

```
rosservice call /yumi/hiqp_kinematics_controller/add_task \
"name: 'jntconfig1'
type: 'TaskJntConfig'
behaviour: ['DynamicsFirstOrder', '2.0']
priority: 1
visibility: 0
parameters: ['0.4', '0.3', '0.3', '0.6', '1.0']"
```

5.2 Joint Velocity Limitation

Being able to hinder the quadratic programming solver to produce certain joint velocity controls (that are later induced on the robot by a lower level controller) is a desirable feat to the user of the framework for a number of

reasons. For instance, for safety reasons one might not want too high joint velocities and simply cutting the resulting controls from the solver risks leading to suboptimal motion generation. Another example could be that the produced controls are simply not achievable by the actual actuators on the robot. By allowing the user to set such velocity limits the solver can take such desired characteristics of the motion generation into account when solving for optimal controls.

In this section we begin first with a straight forward joint velocity limitation task formulation that, as we will see, will be implementable by a joint effort controller. Later we adjust this formulation to better suit a joint velocity controller interface.

A joint velocity limitation task that bounds each joint velocity by $\pm\delta_i$ for each joint i can be written as

$$e = \sum_{i=1}^{n} \left(\dot{q}_i^2 - \delta_i^2 \right) \tag{5.4}$$

whose derivation with respect to time becomes

$$\dot{e} = \frac{d}{dt} \sum_{i=1}^{n} \dot{q}_i^2 = \frac{d}{dt} (\dot{\boldsymbol{q}}^T \dot{\boldsymbol{q}}) = 2\dot{\boldsymbol{q}}^T \ddot{\boldsymbol{q}} \tag{5.5}$$

and we write the task jacobian as

$$J_e = 2\dot{\boldsymbol{q}}^T \tag{5.6}$$

where $J_e \in \mathbb{R}^{1\times n}$. However as J_e in this case maps joint acceleration space to the task function space this formulation is only suitable for a joint effort controller.

Another way of enforcing the quadratic programming solver to limit joint velocities is by customizing the task dynamics. By setting the task function value to equal the joint positions the desired task dynamics can be let to act as a joint velocity limitation. We write

$$\boldsymbol{e} = \boldsymbol{q} \tag{5.7}$$

where $\boldsymbol{e} \in \mathbb{R}^{n\times 1}$. This then leads to the task jacobian being the n-by-n identity matrix. We then set the customized task dynamics as

$$\dot{\boldsymbol{e}}^* = \boldsymbol{\delta} \tag{5.8}$$

where $\boldsymbol{\delta} \in \mathbb{R}^{n\times 1}$ is a vector containing the velocity limits for each joint. When used as a top priority task other tasks will be solved in the null-space of this task, i.e. allowing only velocity controls that lie below or above the limitation $\boldsymbol{\delta}$ depending on the sign of the inequality task.

A sample service call to start a joint velocity limitation task for a 5-joint robot is given below.

Figure 5.2: A sample ros service call to start a joint velocity limitation task.

```
rosservice call /yumi/hiqp_kinematics_controller/add_task \
"name: 'jntlimit1'
type: 'TaskJntLimits'
behaviour: ['DynamicsJntLimits', '0.1', '0.1', '0.1', '0.1',
    '0.1']
priority: 1
visibility: 0
parameters: ['<']"
```

5.3 Geometric Projection

This section describes a class of tasks that attempts to position a geometric point fixed to one link frame relative to a geometric primitive fixed in another link frame. The `TaskGeometricProjection` class extends the `TaskFunction` class with a generic interface to allow for *Point-on-Primitive* projections of end-effector positions. Applying this task to two existing geometric primitives, of which the first must be of type `GeometricPoint`, results in controls that positions the point relative to the other geometric primitive. This class of tasks is generic in nature in that regardless of what primitive the given point is coupled with the calculation of the task function value and the task jacobian is made identically once the geometric projection onto the second primitive has been determined. The following generic instantiations of `TaskGeometricProjection<>` are currently implemented:

- `TaskGeometricProjection<GeometricPoint, GeometricPoint>`
- `TaskGeometricProjection<GeometricPoint, GeometricLine>`
- `TaskGeometricProjection<GeometricPoint, GeometricPlane>`
- `TaskGeometricProjection<GeometricPoint, GeometricBox>`
- `TaskGeometricProjection<GeometricPoint, GeometricCylinder>`
- `TaskGeometricProjection<GeometricPoint, GeometricSphere>`

The part of the task function and jacobian calculations that are common for all combinations of primitives are made in the `apply` method, see Figure **??** for a UML diagram of the `TaskGeometricProjection` class.

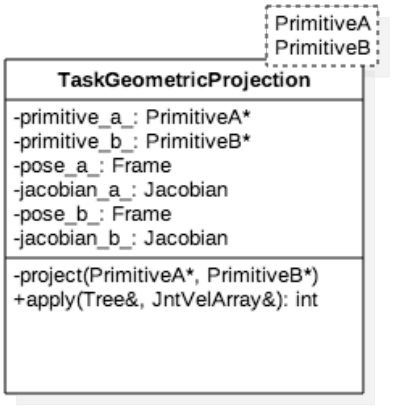


Figure 5.3: UML diagram showing the template class `TaskGeometricProjection`. The Frame, Jacobian, Tree and JntVelArray types are members of KDL.

This involves calculating the jacobian of each of the two link frames that the two end-effector primitives are attached to and fixed in, along with the current poses of those link frames. The jacobians and poses of each link frame are then used by each specialization of the `project` method to compute the final task function value and the task jacobian. We cover the details of these implementations in the oncoming sections.

Since each projection only should constrain the movement of the robot in one dimension, that is movement restricted along the line between one point and its projection on another primitive, each of the primitive combination tasks are designed as one dimensional tasks. Practically this means that the task function is a scalar, and that the task jacobian has size $1 \times n$ where n is the number of joints of the robot.

To further explain the user-interaction with this class of tasks a sample ROS service call to start a point-on-plane equality task is given in Figure **??**.

5.3.1 Point-on-Point

Consider two points $\mathcal{P}1$ and $\mathcal{P}_2$ fixed to each respective link frame F_1 and F_2. Let $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ be vectors in the world frame from the origin to the points. We write these as

$$\boldsymbol{p}_i = \boldsymbol{p}'_i + \boldsymbol{d}_i \tag{5.9}$$

Figure 5.4: Sample ros service call to start a geometric projection task.

```
rosservice call /yumi/hiqp_kinematics_controller/add_task \
"name: 'geomproj1'
type: 'TaskGeometricProjection'
behaviour: ['DynamicsFirstOrder', '10']
priority: 1
visibility: 0
parameters: ['point', 'plane', 'mypoint1 = myplane2']"
```

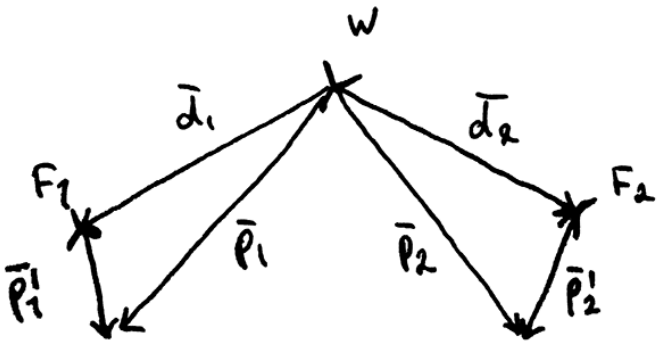


Figure 5.5: Point on point task sketch.

where $\boldsymbol{p}'_i$ is the vector from the origin of frame F_i to the point p_i and $i = 1, 2$. This is depicted in Figure 5.5.

We choose our task function as the square of the euclidean distance between these two points by writing

$$e = (\boldsymbol{p}_2 - \boldsymbol{p}_1)^T(\boldsymbol{p}_2 - \boldsymbol{p}_1) \tag{5.10}$$

By differentiating this task function formulation with respect to time we can derive the task jacobian. We write

$$\dot{e} = 2(\boldsymbol{p}_2 - \boldsymbol{p}_1)^T(\dot{\boldsymbol{p}}_2 - \dot{\boldsymbol{p}}_1) \tag{5.11}$$

For any point formulated as in Equation (5.9) we write its time derivative as

$$\dot{\boldsymbol{p}}_i = \boldsymbol{J}^v_{p_i}\dot{\boldsymbol{q}} \tag{5.12}$$

where $\boldsymbol{J}^v_{p_i} \in \mathbb{R}^{3\times n}$ is the upper half of the jacobian, or equivalently the velocity jacobian, with respect to the point $\boldsymbol{p}_i$, and $i = 1, 2$.

The velocity jacobian of $\boldsymbol{p}_i$ can be expressed as

$$\boldsymbol{J}^v_{p_i} = \begin{bmatrix}\boldsymbol{k}_1 \times (\boldsymbol{r}_1 + \boldsymbol{p}'_i) & \boldsymbol{k}_2 \times (\boldsymbol{r}_2 + \boldsymbol{p}'_i) & \cdots & \boldsymbol{k}_n \times (\boldsymbol{r}_n + \boldsymbol{p}'_i)\end{bmatrix} \tag{5.13}$$

where $\boldsymbol{k}_j$ is the unit vector pointing along the rotation axis of joint j, and $\boldsymbol{r}_j$ is the vector from the origin on link frame j to the origin of the end-effector's link frame. n is the number of joints of the kinematic tree, or equivalently the robot.

The velocity jacobian for a point p_i that is fixed in a frame F_i can thus be written as

$$\begin{gathered}\boldsymbol{J}^v_{p_i} = \boldsymbol{J}^v_{d_i} + \boldsymbol{J}^v_{p'_i} \\ \text{where} \\ \boldsymbol{J}^v_{d_i} = \begin{bmatrix}\boldsymbol{k}_1 \times \boldsymbol{r}_1 & \cdots & \boldsymbol{k}_1 \times \boldsymbol{r}_n\end{bmatrix} \\ \boldsymbol{J}^v_{p'_i} = \begin{bmatrix}\boldsymbol{k}_1 \times \boldsymbol{p}'_i & \cdots & \boldsymbol{k}_n \times \boldsymbol{p}'_i\end{bmatrix}\end{gathered} \tag{5.14}$$

Combining Equations (5.11), (5.12) and (5.14) now yields the task jacobian as

$$\boldsymbol{J}_e = 2(\boldsymbol{p}_2 - \boldsymbol{p}_1)^T(\boldsymbol{J}^v_{d_2} + \boldsymbol{J}^v_{p'_2} - \boldsymbol{J}^v_{d_1} - \boldsymbol{J}^v_{p'_1}) \tag{5.15}$$

and we have that $\boldsymbol{J}_e \in \mathbb{R}^{1\times n}$.

We note that projections of points onto different geometric primitives always can be regarded as point-on-point projections. The difference with an actual point-on-point projection is that the projected point on the other geometric primitive might vary as the robot motion evolves. We will reuse this notion for projection tasks of higher order geometric primitives.

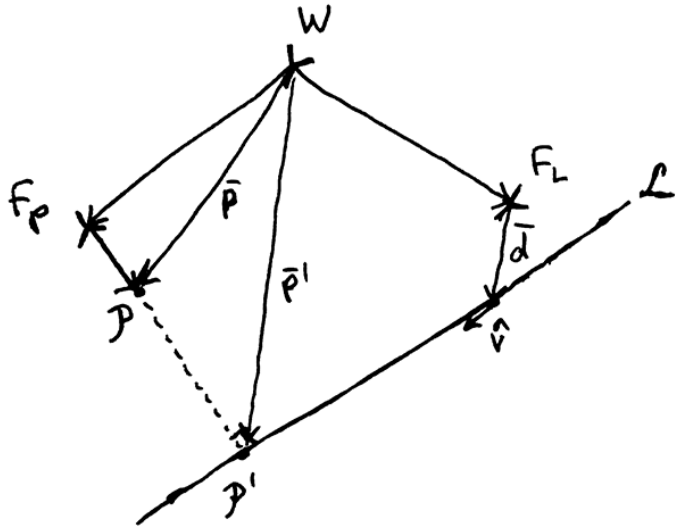


Figure 5.6: Point on line task sketch.

5.3.2 Point-on-Line

To achieve a point-on-line task we recompute at each time step the projection point $\mathcal{P}'$ of the given point $\mathcal{P}$ on the given line $\mathcal{L}$. The points $\mathcal{P}$ and $\mathcal{P}'$ are then regarded as a point-on-point task and we reuse the formulations from the previous section.

To compute a vector in the world frame, $\boldsymbol{p}'$ to the projection point $\mathcal{P}'$ we let $\mathcal{L}$ be a line such that

$$\mathcal{L} = \{\boldsymbol{l} \in \mathbb{R}^3 : \boldsymbol{l} = \boldsymbol{d} + \lambda\hat{\boldsymbol{v}},\ \lambda \in \mathbb{R}\} \tag{5.16}$$

where $\boldsymbol{d} \in \mathbb{R}^3$ is the offset of the line from the origin of its link frame[1], and $\hat{\boldsymbol{v}}$ is a unit vector giving the direction of the line, see the Figure 5.6. The vector $\boldsymbol{p}'$ can then be written as

$$\begin{aligned} \boldsymbol{p}' &= \boldsymbol{d} + \lambda'\hat{\boldsymbol{v}} \\ \lambda' &= (\boldsymbol{p} - \boldsymbol{d})^T\hat{\boldsymbol{v}} \end{aligned} \tag{5.17}$$

where $\boldsymbol{p}$ is the vector to the point $\mathcal{P}$.

We get the task function value and the task jacobian from setting $\boldsymbol{p}_1 = \boldsymbol{p}$ and $\boldsymbol{p}_2 = \boldsymbol{p}'$ in Equations (5.10) and (5.15).

[1] this could be any point on the line

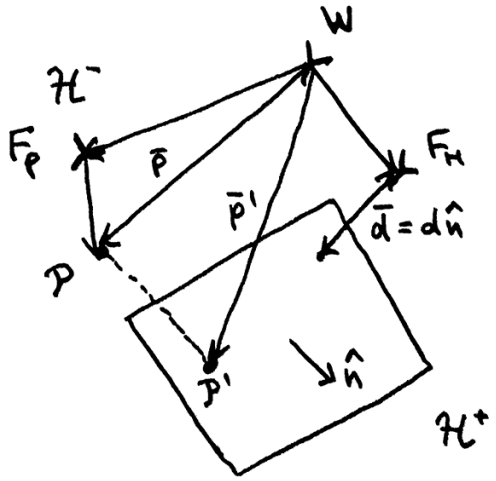


Figure 5.7: Point on line task sketch.

5.3.3 Point-on-Plane

The point-on-plane projection task aims at positioning a point relative to a plane. We denote the point $\mathcal{P}$ fixed to one link frame and the plane $\mathcal{H}$ fixed to another link frame and we distinguish between the *positive* and *negative* half-spaces *under* and *above* the plane. The definition of the plane $\mathcal{H}$ is

$$\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^3 : \hat{\boldsymbol{n}}^T \boldsymbol{x} - d = 0\} \tag{5.18}$$

where $\hat{\boldsymbol{n}}$ is the normal unit vector of the plane and the scalar d is the offset of the plane from the origin of the link frame to which it is attached along $\hat{\boldsymbol{n}}$. See Figure 5.7.

The half-spaces under, $\mathcal{H}^-$, and above, $\mathcal{H}^+$, the plane are determined by the normal direction of the plane and are defined as

$$\begin{aligned} \mathcal{H}^+ &= \{\boldsymbol{x} \in \mathbb{R}^3 : \hat{\boldsymbol{n}}^T \boldsymbol{x} - d \geq 0\} \\ \mathcal{H}^- &= \{\boldsymbol{x} \in \mathbb{R}^3 : \hat{\boldsymbol{n}}^T \boldsymbol{x} - d \leq 0\} \end{aligned} \tag{5.19}$$

A less-than point-on-plane task would try to put the point under the plane, and a greater-than point-on-plane task would try to put it above the plane.

The task function value can now be interpreted as the signed euclidean distance between the point $\mathcal{P}$ and its projection point $\mathcal{P}'$ on the plane $\mathcal{H}$.

Thus we can write

$$e = \hat{\boldsymbol{n}}^T(\boldsymbol{p} - \boldsymbol{d}) + (\boldsymbol{p} - \boldsymbol{d})^T\dot{\boldsymbol{n}} \tag{5.20}$$

where $\boldsymbol{d} = d\hat{\boldsymbol{n}}$. The reason we actually define the vector $\boldsymbol{d}$ is that it is fixed to the frame $\mathcal{F}_{\mathcal{H}}$ and therefore it's jacobian will be calculated in `apply` before using it in the specialized version of `project`. This distance measure is signed to distinguish between the point being under or above the plane.

From its time derivative we can extract the task jacobian. We write

$$\dot{e} = (\hat{\boldsymbol{n}}^T(J_p^v - J_d^v) + (\boldsymbol{p} - \boldsymbol{d})^T J_n^v)\dot{\boldsymbol{q}} \tag{5.21}$$

and

$$J_e = \hat{\boldsymbol{n}}^T(J_p^v - J_d^v) + (\boldsymbol{p} - \boldsymbol{d})^T J_n^v \tag{5.22}$$

where $J_e \in \mathbb{R}^{1\times n}$.

This task could also be written as a point-on-point task by defining a projection point $\mathcal{P}'$ on the plane and then reusing the point-on-point formulation from Section 5.3.1. However, a point-on-point formulation is unable to consider any orientation relative to the plane and we would in that case lose the ability to have the point-on-plane task implemented as an inequality task. Also, the short and concise formulation given in this section motivates a stand-alone solution which we will reuse in the coming sections.

5.3.4 Point-on-Box

The point-on-box projection task regards positioning of a point relative to a six-sided box with orthogonal sides. There are a number of incentives for providing such a geometric projection which we list below.

1. Positioning a point outside of a space enclosed by a box is not applicable by using, for example, six point-on-plane tasks, as keeping the point above all non-parallel planes at the same time is never achievable. Interaction between multiple point-on-plane tasks is therefore necessary, which motivates writing a point-on-box projection task.

2. Using six point-on-plane tasks for keeping a point inside a rectangular space will induce six constraints in the quadratic programming solver which is not necessary as at each time step the desired movement relates to moving the point in a straight line. When multiple such tasks are added to a compound problem using six slack constraints instead of one will affect the speed of the controller. A point-on-box task can, as we show below, be implemented using a scalar task function which therefore will affect performance.

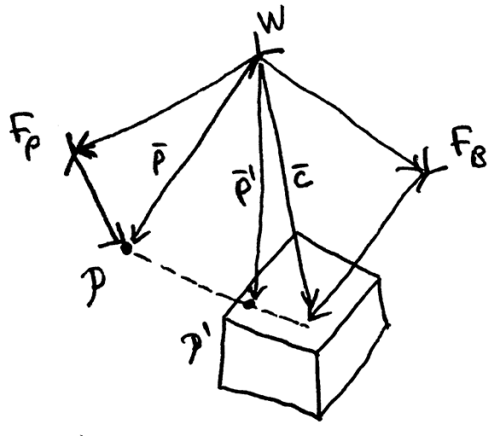


Figure 5.8: Point on box task sketch.

3. By providing a common point-on-box task the user can more easily and more quickly define and parametrize the robot's environment for interaction with it. This type of task is suitable for defining spaces in the environment where certain end-effector points are not allowed to enter, as for obstacle avoidance for instance.

Let the given point be denoted as $\mathcal{P}$, the boundary of the box $\mathcal{B}$ and the projection point of $\mathcal{P}$ on $\mathcal{B}$ as $\mathcal{P}'$. $\mathcal{P}'$ is defined as the point on $\mathcal{B}$ that lies on the line from the box's center to $\mathcal{P}$, regardless of whether $\mathcal{P}$ lies inside or outside the box. We then let $\boldsymbol{p}$ be a vector from the world origin to $\mathcal{P}$, $\boldsymbol{p}'$ a vector to $\mathcal{P}'$, $\boldsymbol{c}$ a vector to the box's center, $\boldsymbol{R}$ the rotation matrix of the box relative to the world frame[2], and $\boldsymbol{S}$ a scaling matrix of the box such that

$$\boldsymbol{S} = Diag(w, d, h) \tag{5.23}$$

where w is the box's width, d its depth and h its height. Please refer to Figure 5.8

By performing a transformation (translate, rotate and scale) on $\boldsymbol{p}$ to an affine space we can derive a vector to the point $\mathcal{P}$ from the box's origin in a frame F that describes the box's geometry as a unit cube. We call this translated, rotated and scaled vector $\boldsymbol{x}$ and write

$$\boldsymbol{x} = \boldsymbol{S}^{-1}\boldsymbol{R}(\boldsymbol{p} - \boldsymbol{c}) \tag{5.24}$$

[2]Note that the box is allowed to have a fixed rotation inside the link frame to which it is attached.

Given that the geometry of the box in the frame F, which $\boldsymbol{x}$ is described in, is a unit cube, each of the six planes encompassing the box's inner space has their normals parallel with one of the frame axes, and all have an offset of 0.5. We can exploit this geometry to easily compute a vector from the center of the box to the projection point $\mathcal{P}'$ in this frame F. By multiplying $\boldsymbol{x}$ with a factor that makes the component of $\boldsymbol{x}$ with the largest absolute value become ± 0.5 we get a vector $\boldsymbol{x}'$ that point from the box's center to $\mathcal{P}'$ in frame F. To achieve this we write

$$\boldsymbol{x}' = \lambda \boldsymbol{x} \tag{5.25}$$

$$\lambda = \frac{1}{2} \cdot \frac{1}{\max(|\boldsymbol{x}^T \hat{\boldsymbol{x}}|,\, |\boldsymbol{x}^T \hat{\boldsymbol{y}}|,\, |\boldsymbol{x}^T \hat{\boldsymbol{z}}|)} = \frac{1}{2} \cdot \frac{1}{||\boldsymbol{x}||_\infty} \tag{5.26}$$

By re-scaling, re-rotating and re-translating we can now get the vector $\boldsymbol{p}'$ going from the origin of the world frame to the projection point $\mathcal{P}'$ by writing

$$\boldsymbol{p}' = \boldsymbol{R}^{-1} \boldsymbol{S} \boldsymbol{x}' + \boldsymbol{c} \tag{5.27}$$

To formulate the task function and the task jacobian we reuse the formulations from the point-on-plane task. We defining a plane $\mathcal{K}$ as

$$\begin{gathered} \mathcal{K} = \{\boldsymbol{x} \in \mathbb{R}^3 : \hat{\boldsymbol{n}}^T \boldsymbol{x} - d = 0\} \\ \hat{\boldsymbol{n}} = \frac{\boldsymbol{p} - \boldsymbol{p}'}{|\boldsymbol{p} - \boldsymbol{p}'|} \\ d = \hat{\boldsymbol{n}}^T \boldsymbol{p}' \\ \boldsymbol{d} = d\hat{\boldsymbol{n}} \end{gathered} \tag{5.28}$$

The task function and task jacobian for the point-on-box task are then the same as in Equation (5.20) and (5.22) with parameters as in Equation (5.28).

5.3.5 Point-on-Cylinder

A task function that places a point relative to the surface of an open cylinder is geometrically very similar to that of a point-on-line task. While a point-on-line task can not be distinguished between an inequality interpretation and an equality interpretation, a point-on-cylinder can however. A less-than-or-equal-to point-on-cylinder task would try to achieve positioning the point somewhere inside the cylinder, while a greater-than-or-equal-to task would position in outside the cylinder, and in the equality case position it on the cylinders surface. To achieve this, the task function formulation from the point-on-line task must be extended with a term related to the cylinder's radius. As the point-on-line task function value is

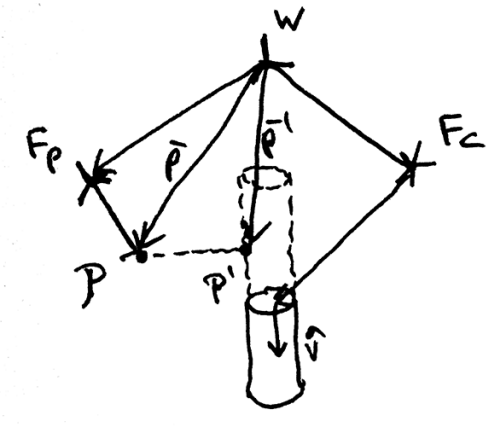


Figure 5.9: Point-on-cylinder task sketch.

an always positive scalar (the shortest euclidean distance to a point the line squared), subtracting the cylinder's radius squared would achieve this. We also note that such an appendix to the task function value is never time dependent, and therefore leaves the task jacobian unchanged. We will now define the point-on-cylinder problem and its solution.

Let the cylinder be defined by an offset vector $\boldsymbol{d}$, a unit vector going in the direction of the main axis of the cylinder $\hat{\boldsymbol{v}}$ and a radius scalar r. The cylinder is regarded as open. The task function value is then defined as the difference between the point-on-line task function value and r^2. Please see Figure 5.9.

We write the task function as

$$e = (\boldsymbol{p} - \boldsymbol{p}')^T(\boldsymbol{p} - \boldsymbol{p}') - r^2 \tag{5.29}$$

The expression for the task jacobian is equal to that of the point-on-line task, see Section 5.3.2.

5.3.6 Point-on-Sphere

The point-on-sphere task is a point-on-point task extended with a radius and the ability to be distinguished between an inequality and an equality task. Similar to the formulation of the point-on-line task this extension only affects the task function value in Equation (5.10) with the addition of a term, but not the jacobian in Equation (5.15) as the added term is constant with respect to time.

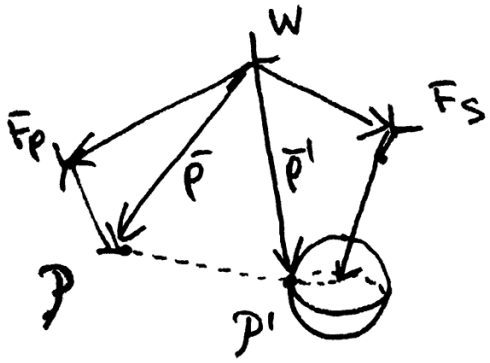


Figure 5.10: Point-on-sphere task sketch.

Consider a sphere with its center at point $\mathcal{C}$ and radius $r \in \mathbb{R}$, and a point $\mathcal{P}$. Consider also two vectors $\boldsymbol{c}$ going from the world origin to the point $\mathcal{C}$, and $\boldsymbol{p}$ going from the world origin to the point $\mathcal{P}$. Please see Figure 5.10. The task function value then becomes

$$e = (\boldsymbol{p} - \boldsymbol{c})^T(\boldsymbol{p} - \boldsymbol{c}) - r^2 \tag{5.30}$$

and the task jacobian remains unchanged compared to Equation (5.15) with $\boldsymbol{p}_1 = \boldsymbol{p}$ and $\boldsymbol{p}_2 = \boldsymbol{c}$.

5.4 Geometric Alignment

While the positioning of end-effectors is an important feature when it comes to robotic manipulation, simply placing an end-effector at a certain position is not enough to ensure proper interaction with the environment. Consider for instance, as in the case of the YuMi robot, picking up a box with one of the grippers. If the gripper in not directed towards one of the sides of the box, the gripper could be grasping the box's edges or corners instead of its sides which would result in an unstable lift of the box. This section therefore covers basic alignment task definitions and their implementations.

We regard first a general task formulation for aligning one line to be parallel with another line while allowing for a certain degree of alignment error. This is not to be confused with positioning a line to intersect with another line which is another type of task. For example both grippers of the YuMi robot can with a line-with-line alignment task be made to

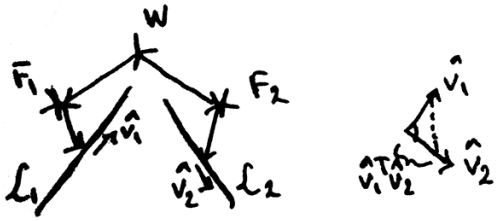


Figure 5.11: line with line task sketch.

always point in the same direction, or to always be opposite each other when executing for example two-hand grasping.

The following spcializations of `TaskGeometricAlignment` have been implemented:

- `TaskGeometricAlignment<GeometricLine, GeometricLine>`
- `TaskGeometricAlignment<GeometricLine, GeometricPlane>`
- `TaskGeometricAlignment<GeometricLine, GeometricCylinder>`
- `TaskGeometricAlignment<GeometricLine, GeometricSphere>`

5.4.1 Line-with-Line Alignment

Consider two lines $\mathcal{L}_1$ and $\mathcal{L}_2$ fixed in two end-effector frames F_i and described in the world frame, each with their offset vector $\boldsymbol{d}_i$ and normal directional vector $\hat{\boldsymbol{v}}_i$ for $i = 1, 2$ as

$$\begin{gathered}\mathcal{L}_i = \{\boldsymbol{l}_i \in \mathbb{R}^3 : \boldsymbol{l}_i = \boldsymbol{c}_i + \boldsymbol{d}_i + \lambda_i \hat{\boldsymbol{v}}_i,\ \lambda_i \in \mathbb{R}\} \\ i = 1, 2\end{gathered} \tag{5.31}$$

where $\boldsymbol{c}_i$ is the origin of frame F_i described in the world frame, see Figure 5.11. The offset and directional vectors are here expressed in the world frame but fixed in the link frame of each end-effector F_i. A task formulation that leads to aligning the two lines can be written as

$$e = \hat{\boldsymbol{v}}_1^T \hat{\boldsymbol{v}}_2 - cos(\delta) \tag{5.32}$$

In other words, the cosine of the angle between the two unit vectors minus the cosine of the angular error margin δ. We chose this definition since it makes it easy to find the jacobian of this expression since δ is constant. This definition of the angle computes a rotation around the vector $\hat{\boldsymbol{v}}_1 \times \hat{\boldsymbol{v}}_2$. The constant term δ enables the use of inequality tasks to ensure that the

angle between two line stays larger or smaller than δ, and equality tasks that the angle equals δ. Geometrically this can be likened to placing a cone around one of the lines and keeping the other line inside, on, or outside the cone. However, this comparison is not fully valid as the task only regards the rotation of each line and not their relative position, i.e. the task will not ensure that the two lines will intersect.

From derivation of the task function with respect to time we get

$$\dot{e} = \dot{\hat{\boldsymbol{v}}}_1^T \hat{\boldsymbol{v}}_2 + \hat{\boldsymbol{v}}_1^T \dot{\hat{\boldsymbol{v}}}_2 \tag{5.33}$$

Essentially, this projects the angular velocity of the end-effector frame onto the desired axis of rotation. We can now see that

$$\hat{\boldsymbol{v}}_1^T \dot{\hat{\boldsymbol{v}}}_2 = \hat{\boldsymbol{v}}_1^T \left[\boldsymbol{k}_1 \times \boldsymbol{v}_2 \cdots \boldsymbol{k}_n \times \boldsymbol{v}_2\right] \dot{\boldsymbol{q}} = (\hat{\boldsymbol{v}}_1 \times \hat{\boldsymbol{v}}_2)^T J_{v_2}^{\omega} \dot{\boldsymbol{q}} \tag{5.34}$$

We can now write the task jacobian as

$$\begin{gathered} J_e = (\hat{\boldsymbol{v}}_1 \times \hat{\boldsymbol{v}}_2)^T J_{v_2}^{\omega} + (\hat{\boldsymbol{v}}_2 \times \hat{\boldsymbol{v}}_1)^T J_{v_1}^{\omega} = \\ (\hat{\boldsymbol{v}}_1 \times \hat{\boldsymbol{v}}_2)^T (J_{v_2}^{\omega} - J_{v_1}^{\omega}) \end{gathered} \tag{5.35}$$

where $J_e \in \mathbb{R}^{1 \times n}$.

5.4.2 Line Alignment with Other Primitives

The line-with-line alignment task formulation can easily be extended to encompass line-with-plane, line-with-box, line-with-cylinder and line-with-sphere, while a point on the other hand never has any orientation and cannot be aligned with. Aligning a line with a plane naturally has two apparent interpretations: either the line is aligned to be parallel with or perpendicular to the plane. Both these interpretations would be useful for applications in real control implementations, however as we extend the definition to encompass also boxes, cylinders and spheres the parallel alignment is not longer as applicable. Letting a line be parallel to the surface of a cylinder or a sphere generally results in no condition on the orientation of the line at all, since all possible line orientations have tangential representations somewhere on the cylinder and on the sphere. Instead, letting line alignment with a plane, box, cylinder or sphere signify the line being perpendicular to the surface at the point of intersection between the line and the other geometric primitive yields a more clearly defined alignment behaviour and thus renders more useful for robotic control applications.

For a line-with-plane alignment task this would involve having the line always be parallel with the planes normal vector. This is then very similar to the line-with-line alignment formulation which we reuse in this case.

For a line-with-cylinder and line-with-sphere alignment tasks we find the projection $\mathcal{P}'$ on the cylinder/sphere of the point $\mathcal{P}$ given by the offset vector of the line. The line's directional vector is then aligned with the normal vector or the cylinder/sphere at $\mathcal{P}'$. This line can be placed at, for instance, the center of a gripper and point in the approach direction of the gripper. By aligning this line with a plane, cylinder or sphere the gripper will always be directed perpendicular to the surface at the point of contact. This is a more useful interpretation of alignment to real world applications than letting a line be parallel with the surface at point $\mathcal{P}'$.

This definition does not hold for line-with-box alignment however. As the projection point $\mathcal{P}'$ can end up on one or the box's edges or corners, the choice of normal vector becomes ambiguous. We have therefore chosen not to implement this task.

5.5 First-Order Task Dynamics

A common type of task dynamics is exponential decay. The main reason for this is that exponential decay does not depend on time. We have implemented first-order exponential decay according to Equation 5.36. This behaviour is indirectly evaluated in the gripping experiment, see Chapter 6, where it is being used to perform gripping motions.

$$\dot{\boldsymbol{e}}^* = -\lambda \boldsymbol{e} \tag{5.36}$$

5.6 Minimal Jerk Task Dynamics

A minimal jerk task dynamics tries to fulfill a task while considering the task evolution to minimize the total jerk on the kinematic structure throughout the execution of the task. While this is used in applications to minimize total energy spent in execution or minimizing the strain on mechanical parts for instance, that would imply a minimal jerk evolution of each joint. As tasks are often defined in a lesser dimensional space than the configuration space, the definition of minimal jerk with respect to each joint's velocity is not fully applicable. We can, however, define a minimal jerk task behaviour in the task space by using the same mathematical definition of minimal jerk evolution as given by [9]. We define the minimal task dynamics as in Equation (5.37).

$$\begin{aligned} \boldsymbol{e}^*(t) &= \boldsymbol{e}(t_0) + (\boldsymbol{e}(t_f) - \boldsymbol{e}(t_0)) \left(10\tau^3 - 15\tau^4 + 6\tau^5\right) \\ \text{where} \quad \tau &= \frac{t - t_0}{t_f - t_0} \end{aligned} \tag{5.37}$$

As we desire to define a reference task behaviour $\dot{\boldsymbol{e}}^*$, we derive the expression in Equation (5.37) with respect to time and get

$$\dot{\boldsymbol{e}}^*(t) = -\frac{30}{t_f - t_0}\left(\boldsymbol{e}(t_f) - \boldsymbol{e}(t_0)\right)\left(\tau^2 - 2\tau^3 + \tau^4\right) \tag{5.38}$$

This notation relies on knowing the initial and final task function values. The parameter $t_f - t_0$ can be seen as a design parameter (the total task duration) as the initial time is the system time at the initiation instant of the task. In many cases the final task value is zero, however to allow the user to customize his/her own tasks our implementation the term $\boldsymbol{e}(t_f)$ is kept in the task dynamics definition in Equation (5.38).

From experience of applying the task dynamics in Equation (5.38) we saw that the task fulfillment rate (the final task function value divided by the initial task function value) was not satisfactorily low[3]. We therefore introduced a feed-forward P-controller that penalizes deviations from the minimal jerk task trajectory given in Equation (5.37). The resulting desired reference behaviour then instead becomes

$$\begin{gathered}
\dot{\boldsymbol{e}}^*(t) = \dot{\boldsymbol{e}}^*_{mj}(t) + \dot{\boldsymbol{e}}^*_P(t) \\
\text{where} \\
\dot{\boldsymbol{e}}^*_{mj}(t) = -\frac{30}{t_f - t_0}\left(\boldsymbol{e}(t_f) - \boldsymbol{e}(t_0)\right)\left(\tau^2 - 2\tau^3 + \tau^4\right) \\
\dot{\boldsymbol{e}}^*_P(t) = -K(\boldsymbol{e}(t) - \boldsymbol{e}^*(t))
\end{gathered} \tag{5.39}$$

where $\boldsymbol{e}^*(t)$ is the desired task function value when applying minimal jerk task dynamics, i.e. the value of $\boldsymbol{e}^*(t)$ in Equation (5.37). In effect the feed-forward P-controller becomes a First-Order Task Dynamics where the reference value is the minimal jerk task trajectory instead of zero as it is defaulted to in our First-Order Task Dynamics implementation, see Section 5.5.

The ROS service-call used to add a task using minimal jerk dynamics is shows in Listing 5.12.

[3]For the tasks we tested the ratio was above 4%.

Figure 5.12: Sample ros service call of a minimal jerk dynamics. Design parameters are $t_f = 3.4$ seconds and $k = 35$.

```
rosservice call /yumi/hiqp_kinematics_controller/add_task \
"name: 'taskname'
type: 'TaskType'
behaviour: ['DynamicsMinimalJerk', '3.4', '35']
priority: 1
visibility: 0
parameters: ['task parameters']"
```

Chapter 6

Evaluation

To evaluate the HiQP control framework we look closer at the performance of gripping motion generations, and minimal jerk task dynamics.

6.1 Software Quality

There are various design measures used to evaluate the quality of a software design [7]. Many of these correlate with, for instance, fault-proneness and thus actually span a lower dimension metric space for code design quality than the total number of measures. That, along with the fact that many measures require code analysis software that were not at hand during the project, we looked for a smaller set of design measures. This more concise still comprehensive set of design measures that formed the basis of our evaluation were proposed by [2] and are quoted here in Table 6.1. The work in [2] also links these design metrics to the quality attributes we stated in Chapter 4, see Table 6.4.

The design metrics are only applicable when compared to an earlier version of the code base, or with another library with the same functionality. The number of classes, for example, is relative to the size of the project and to what functionality it shall provide. However, we can regard the design size as the number of classes and the total number of methods to make comparisons between code bases.

In [2] the design metrics of the Microsoft Foundation Classes are given. Although we cannot compare the design size directly as the libraries are of different sizes altogether, we can make some rational comparisons. By dividing the metrics NOH, ANA and DCC, which are related to the design size in number of classes, with DSC we can compare HiQP and MFC. By dividing the metrics NOP and CIS, which are metrics related to the total

Table 6.1: Design Metric Descriptions. This table is quoted from [2].

METRIC	**NAME**	**DESCRIPTION**
DSC	Design Size in Classes	This metric is a count of the total number of classes in the design.
NOH	Number of Hierarchies	This metric is a count of the number of class hierarchies in the design.
ANA	Average Number of Ancestors	This metric value signifies the average number of classes from which a class inherits information. It is computed by determining the number of classes along all paths from the "root" class(es) to all classes in an inheritance structure.
DAM	Data Access Metric	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value of DAM is desired. (Range 0 to 1)
DCC	Direct Class Coupling	This metric is a count of the different classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CAM	Cohesion Among Methods of Class	This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. A metric value close to 1.0 is preferred. (Range 0 to 1)
MFA	Measure of Functional Abstraction	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOP	Number of Polymorphic Methods	This metric is a count of the methods that can exhibit polymorphic behaviour. Such methods in C++ are marked as virtual.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class.
NOM	Number of Methods	This metric is a count of all the methods defined in a class.

Table 6.2: Design Metrics of the HiQP framework.

METRIC	DESIGN PROPERTY	VALUE
DSC	Design Size	30
NOH	Hierarchies	5
ANA	Abstraction	0.53
DAM	Encapsulation	0.93
DCC	Coupling	3.03
CAM	Cohesion	0.56
MFA	Inheritance	0.51
NOP	Polymorphism	2.57
CIS	Messaging	10.59
NOM	Complexity	11.48

number of methods, with NOM we can compare HiQP and MFC for these design measures as well. Also three of the metrics, namely DAM, CAM and MFA, are ratios ranging between 0 and 1 which we compare directly between the two libraries. It is difficult to say whether one of these ratios is good or bad in general, but they work well for telling what library excels over the other on that design property. Therefore the numbers can not be interpreted on their own, but only in comparison.

We make the following observations of the design metrics when comparing HiQP and MFC:

- **Hierarchies**. The ratio NOH/DSC[1] for the HiQP framework is 0.1667 and the maximum for 5 different versions of MFC is 0.0258. From this we conclude that our number of hierarchies in HiQP is relatively higher than that of the MFC.
- **Abstraction**. The ratio ANA/DSC for HiQP is 0.0177 and the highest value for MFC is 0.0233.
- **Coupling**. The ratio DCC/DSC for HiQP is 0.101 and the maximum for MFC is 0.0838 (MFC v.1.0) which indicates that our code involves more coupling than MFC.
- **Polymorphism**. Looking at the measure of polymorphism we can see that the ratio NOP/NOM[2] is 0.2239 for HiQP and the maximum is 0.19 for MFC (v.5.0). That indicates that our code is slightly more polymorphic than MFC.

[1] the number of hierarchies over the total number of classes
[2] the number of polymorphic methods over the total number of methods

Table 6.3: Comparison between the DAM, CAM and MFA design metrics between HiQP and the maximum value for MFC.

	HiQP	**MFC**
DAM	0.93	0.61
CAM	0.56	0.2
MFA	0.51	0.5

Table 6.4: Linkage of design metrics to quality attributes. The sign sifgnifies whether the metric has a positive/negative impact on the quality attribute. Re. = Reusability, Fl. = Flexibility, Un. = Understandability, Fu. = Functionality, Ex. = Extendability, Ef. = Effectiveness.

	Reusa.	**Flexi.**	**Under.**	**Funct.**	**Exten.**	**Effec.**
Design Size	+		+			
Hierarchies		+				
Abstraction			-		+	+
Encapsulation		+	+			+
Coupling	-	-	-		-	
Cohesion	+		+	+		
Inheritance					+	+
Polymorphism		+	-	+	+	+
Messaging	+			+		
Complexity			-			

- **Messaging**. The ratio CIS/NOM is for HiQP 0.9225 and the maximum for MFC is 0.7894.

Regarding the DAM, CAM and MFA ratio-based measures we compare the values of HiQP and the maximum value among the MFC versions in [2] in Table 6.3. Since the DSC and NOM metrics are regarded as metrics of the project size and used to normalize the other metrics we cannot make any comparison using these.

From these measures and their impact on quality attributes given in Table 6.4 we can attempt drawing conclusions on the quality of our code base compared to the MFC library. Regarding *reusability*, as both cohesion and messaging is estimated to be higher in HiQP than in MFC which indicates more reusable code. However, our code is indicating more coupling than the MFC code which decreases the reusability. The same conclusion can be drawn for code *flexibility* as the hierarchies, encapsulation and polymorphism metrics are higher which makes HiQP a more flexible code base than MFC. Although again, the high coupling in HiQP indicates low flexibility. Finally there are more arguments that points

at the HiQP framework being more flexible than MFC than arguments against it. The *understandability* of the code is a more unsure case as the three measures of abstraction, encapsulation and cohesion indicates that the HiQP framework is more understandable in terms of its source code than MFC. On the other hand, both the high coupling and high polymorphism values relative to MFC pulls down the understandability measure for HiQP in comparison with MFC. HiQP can quite easily be said to be more *functional* than MFC since all the three measures it is positively linked to are higher for HiQP than MFC. HiQP is however less *extendisble* as two measures are playing in MFC's favour, namely abstraction and coupling, while the measure of inheritance seems to be roughly the same in both frameworks and only the polymorphism metric is in favour of HiQP. The *effectiveness* is, along with the understandability, difficult to address to the one framework over the other. Abstraction if lower for HiQP which indicates it being less effective, while encapsulation is higher, inheritance being roughly the same and polymorphism only slightly larger. It is therefore difficult to say which of the two frameworks is the most effective. The conclusions on the quality attributes for HiQP and MFC in comparison are summarised in Table 6.5.

Table 6.5

	HiQP	**MFC**
Reusability	+	
Flexibility	+	
Understandability	-	-
Functionality	++	
Extendability		+
Effectivity	-	-

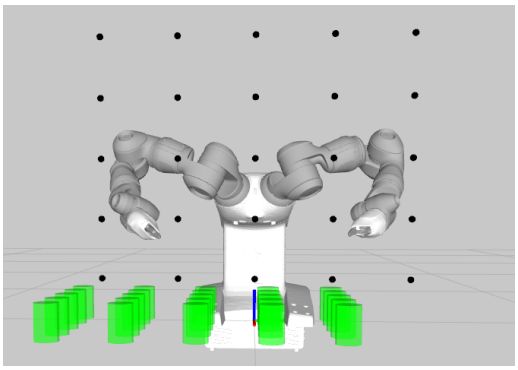
6.2 Gripping

With the HiQP framework being subject to deficiencies associated with local optimal control[3] we wanted to evaluate this approach in a gripping setting. We placed a virtual cylinder at 25 different positions in front of YuMi and set up tasks to initiate grabbing it from 25 different starting positions. The orientation of the wrist was unconstrained for each starting position. The setup is shown in detail in Figure 6.1, however of the 25

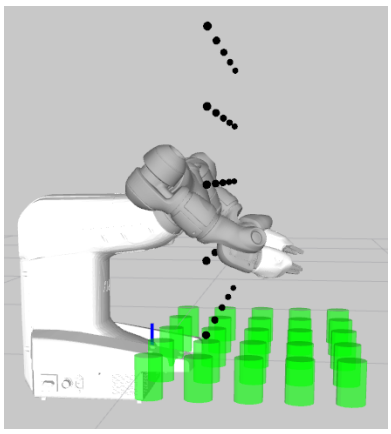
[3]Local optimal control in a temporal sense does not take into account future state solutions and therefore might not find a valid path even if there is one.

intended starting positions one could not be reached by the gripper and was therefore excluded from the experiment.

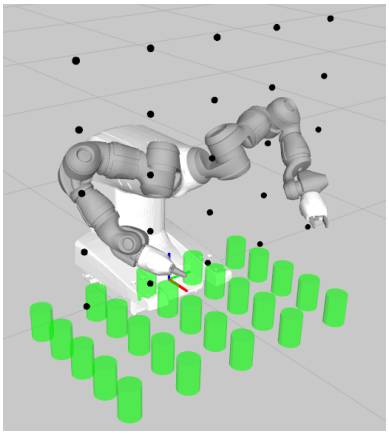
Figure 6.1: Overview of starting positions and cylinders in the gripping experiment.



(a) Front view.



(b) Side view.



(c) Isometric view.

6.2.1 Test Setup

The test was setup to bring the right hand gripper of YuMi from 25 different starting positions to end positions that would indicate successful placement of the gripper to pickup a cylindrical object. This was under the assumption that closing the gripper once all tasks were satisfied yielded

a successful grasp. There were 25 different positions of the cylinder. The starting positions were placed across a grid in the y-z-plane with fixed $x = 0.2$. The cylinder had a radius of 0.05 and was placed across a grid in the x-y-plane coaxially with the z-axis. The cylinder was encapsulated by two planes referred to as the floor plane, at height $z = 0.115$, and the top plane, at height $z = 0.215$. We refer to Figure 6.1 for an overview of the total setup.

The gripping task that was tested consisted of the following subtasks:

1. `bring_gripper_point_to_cylinder` A point-on-cylinder equlity task for the gripper point and the cylinder.
2. `bring_gripper_point_above_floor` A point-on-plane inequality task for the gripper point and the floor plane.
3. `bring_gripper_point_under_plane` A point-on-plane inequality task for the gripper point and the plane just above the cylinder.
4. `align_gripper_with_floor` A line-with-plane alignment task for a line going vertically trough the gripper and the floor plane.
5. `align_gripper_with_cylinder` A line-with-cylinder alignment task for a line going in the gripping direction of the gripper and the cylinder.

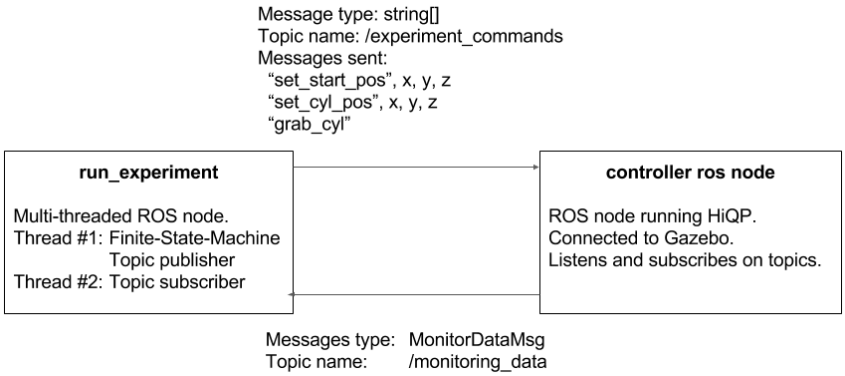


Figure 6.2: Communication diagram for testing gripping.

We wrote a separate multi-threaded program, called `run_experiment`, registered as a ROS node to handle the evolution and recording of the test, see Figure 6.2. `run_experiment` sends a string array message to a

topic subscriber in the controller ROS node which is there parsed. The following list shows the messages parsed and their resulting behaviour of the controller.

- `"set_start_pos, x, y, z"` deactivates all tasks, sets the start point primitive to `(x,y,z)`, activates the `bring_back_to_start` task.
- `"set_cyl_pos, x, y, z"` sets the position of the cylinder to `(x,y,z)`.
- `"grab_cyl"` deactivates all tasks, and activates the tasks involved in grabbing the cylinder.

The `run_experiment` ROS node implemented a finite state machine that cycles through these messages to bring the gripper to one of the 25 starting positions and towards gripping the cylinder at its 25 different locations. `run_experiment` also listens to the `monitoring_data` topic and registers whether the task of gripping the cylinder has been successful or not within 10 seconds from launching it. The gripping of a cylinder was considered successful when all of the tasks involved in grabbing the cylinder had task function value deviations less than 0.01 from 0.

6.2.2 Results

We collected data on whether grabbing the cylinder at one of its 25 designated locations from one of the 24 starting positions of the grabber could be successfully achieved by the controller or not. In Table 6.6 we present the percentage of cylinders that was successfully grabbed from each of the 24 starting positions. In Table 6.7 we show the percentage of starting positions from which each of the 25 positions of the cylinder could successfully be grabbed.

From our results we can see that the non-global solution mechanism in HiQP have a significant effect on whether a set of tasks can be successfully performed or not. On average only 55% of the total 600 test cases were successful. Obvious reasons for this are either that the controller gets stuck at a local minima[4], or that there actually does not exist a valid solution i.e. that the cylinder is not reachable by the gripper. However, from the design of the test cases we know that all cylinders and 24 out of 25 of the starting positions were reachable.

[4]The iterative instantaneous optimization carried out by HiQP is not able to foresee future unsatisfied constraints and is not able to proactively avoid them.

Table 6.6: Map of **starting positions** from which a percentage of the 25 cylinders was successfully grabbed. The mean-row and mean-column shows the average success rate for each row and column, and the value in the down-right cell shows the total success rate as an average of all test cases. *z-positions are written horizontally, and y-positions vertically.*

	0.250	**0.425**	**0.600**	**0.775**	**0.950**	**mean**
-0.450	64%	32%	16%	88%	80%	56%
-0.225	96%	36%	32%	72%	68%	61%
0.0	36%	68%	100%	64%	48%	63%
0.225	60%	56%	48%	60%	36%	52%
0.450	N/A	33%	44%	67%	20%	41%
mean	64%	45%	48%	70%	50%	55%

Table 6.7: Map of **cylinder positions** which a percentage of the 25 starting positions was successfully grabbed from. The mean-row and mean-column shows the average success rate for each row and column, and the value in the down-right cell shows the total success rate as an average of all test cases. *x-positions are written horizontally, and y-positions vertically.*

	0.077	**0.191**	**0.305**	**0.419**	**0.533**	**mean**
-0.533	35%	30%	40%	45%	30%	36%
-0.3415	55%	55%	55%	45%	40%	50%
-0.15	65%	75%	83%	91%	65%	76%
0.0415	40%	80%	75%	91%	65%	70%
0.233	45%	45%	63%	54%	42%	50%
mean	48%	57%	63%	65%	48%	56%

6.3 Minimal Jerk

We evaluated the minimal jerk task dynamics too see how well it performed in comparison to the analytic version. An analytic expression for a cost measure was calculated and its value was also numerically extracted from simulations. We have compared the sum of deviations from the analytic jerk of a generated point-on-point motion, and also the rate at which the task was fulfilled. By writing the rate we mean the final task function value divided by the initial task function value. This section covers the details of this evaluation.

6.3.1 Analytic Cost Function

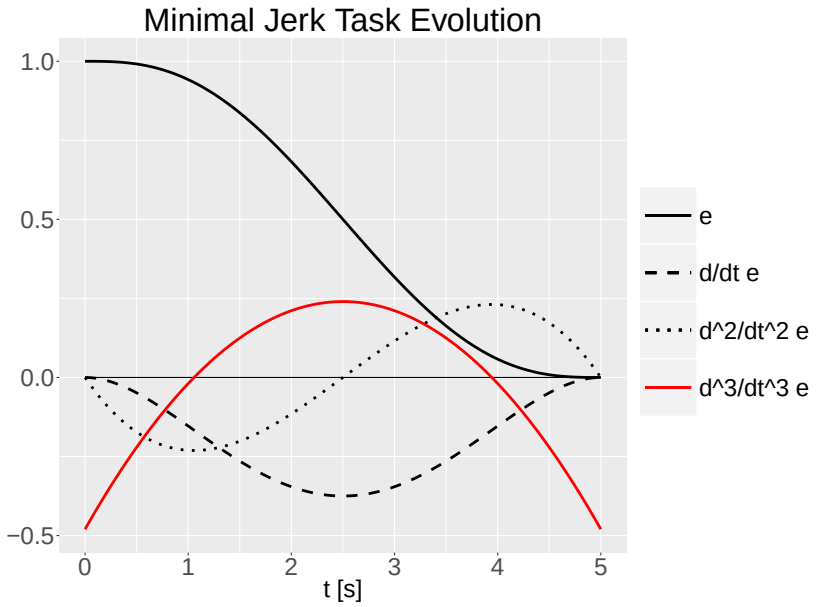
From the definition of minimal jerk found by [9], see Equation 6.1, we compute an analytic expression for the sum of the absolute value of the jerk throughout the task evolution.

$$\begin{aligned} \boldsymbol{e} &= \boldsymbol{e}_0 + (\boldsymbol{e}_f - \boldsymbol{e}_0)(10\tau^3 - 15\tau^4 + 6\tau^5) \\ \text{where } \tau &= \frac{t - t_0}{t_f - t_0} \end{aligned} \tag{6.1}$$

First we calculate the derivatives d/dt, d^2/dt^2, d^3/dt^3 of $\boldsymbol{e}$ as in (6.2), see Figure 6.3 for plots of these functions.

$$\begin{aligned} \frac{d\boldsymbol{e}}{dt} &= \frac{30(\boldsymbol{e}_f - \boldsymbol{e}_0)}{t_f - t_0}(\tau^2 - 2\tau^3 + \tau^4) \\ \frac{d^2\boldsymbol{e}}{dt^2} &= \frac{30(\boldsymbol{e}_f - \boldsymbol{e}_0)}{(t_f - t_0)^2}(2\tau - 6\tau^2 + 4\tau^3) \\ \frac{d^3\boldsymbol{e}}{dt^3} &= \frac{60(\boldsymbol{e}_f - \boldsymbol{e}_0)}{(t_f - t_0)^3}(1 - 6\tau + 6\tau^2) \end{aligned} \tag{6.2}$$

Figure 6.3: Minimal Jerk Task Evolution.



From the expression of jerk in Equation 6.2 we form the sum of the absolute value of the jerk called C, see Equation 6.3. For simplicity we only regard a one dimensional task function e.

$$\begin{gathered} C = \int_{t_0}^{t_f} \left| k(6\tau^2 - 6\tau + 1) \right| dt \\ \text{where } k = \frac{60(e_f - e_0)}{(t_f - t_0)^3} \\ 0 \leq t_0 < t_f, \;\; 0 \leq e_f < e_0 \\ 0 \leq k < \infty \end{gathered} \tag{6.3}$$

We let $t_0 = 0$ for simplicity, and change the integration variable to $\tau = t/t_f$. We write

$$C = k \int_0^1 \left| (6\tau^2 - 6\tau + 1) \right| d\tau \tag{6.4}$$

and

$$\begin{gathered} 6\tau^2 - 6\tau + 1 = 0 \Leftrightarrow \tau = \frac{1}{2}(1 \pm \frac{1}{\sqrt{3}}) \\ \text{let } \tau_1 = \frac{1}{2}(1 - \frac{1}{\sqrt{3}}), \;\; \tau_2 = \frac{1}{2}(1 + \frac{1}{\sqrt{3}}) \end{gathered} \tag{6.5}$$

and from looking at the plot of the jerk in Figure 6.3 and the parts where it is positive/negative specifically, we can now write C as

$$\begin{gathered} C = k \left(- \int_0^{\tau_1} l(\tau) d\tau + \int_{\tau_1}^{\tau_2} l(\tau) d\tau - \int_{\tau_2}^{1} l(\tau) d\tau \right) \\ \text{where } l(\tau) = 6\tau^2 - 6\tau + 1 \end{gathered} \tag{6.6}$$

and we have

$$\begin{gathered} C = k \left([L(\tau)]_{\tau_1}^{0} + [L(\tau)]_{\tau_1}^{\tau_2} + [L(\tau)]_{1}^{\tau_2} \right) \\ \text{where } L(\tau) = 2\tau^3 - 3\tau^2 + \tau \end{gathered} \tag{6.7}$$

and we develop the expression by writing

$$\begin{gathered} C = k(-2L(\tau_1) + 2L(\tau_2) - L(1)) \\ = k \left(2(L(\tau_2) - L(\tau_1)) - (2 - 3 + 1) \right) \end{gathered} \tag{6.8}$$

where

$$
\begin{aligned}
& L(\tau_2) - L(\tau_1) \\
& = \\
& 2(\frac{1+1/\sqrt{3}}{2})^3 - 3(\frac{1+1/\sqrt{3}}{2})^2 + \frac{1+1/\sqrt{3}}{2} \\
& -2(\frac{1-1/\sqrt{3}}{2})^3 + 3(\frac{1-1/\sqrt{3}}{2})^2 - \frac{1-1/\sqrt{3}}{2} \\
& = \\
& \frac{1}{4}\left((1+\frac{1}{\sqrt{3}})^3 - (1-\frac{1}{\sqrt{3}})^3 - (1+\sqrt{3})^2 + (-1+\sqrt{3})^2\right) + \frac{1}{\sqrt{3}} \\
& = \\
& \frac{1}{4}\left(\frac{3\cdot 3\cdot 2+2}{3\sqrt{3}} - \frac{4\cdot 3}{\sqrt{3}}\right) + \frac{1}{\sqrt{3}} \\
& = \\
& -\left(\frac{1}{\sqrt{3}}\right)^3
\end{aligned}
\tag{6.9}
$$

and from Equations 6.3, 6.8, 6.9 we get C as

$$C = -\frac{40}{\sqrt{3}}\frac{e_f - e_0}{t_f^3} \tag{6.10}$$

When evaluating the performance of our minimal jerk task dynamics implementations we will compare the total sum of the jerk across the motion from simulation with the value of C.

6.3.2 Testbed Setup

To evaluate our implementation of minimal jerk task dynamics we setup a point-on-point equality task to bring two points, attached to each of the two grippers, together. In our test we varied the frequency of the controller as we expected the deviation from the analytic case would decrease with higher frequency. We also varied the total time duration of the motion. Since our implementation allows for adding a feed-forward P-controller we also vary the gain K of this controller, see Section 5.6.

By logging all task jacobian and joint velocity values throughout the motion we used the finite difference coefficient method, see [10], to numerically derive the jerk of the task function value. Technically, the values of the task jacobian $\boldsymbol{J}$ and the generated controls $\dot{\boldsymbol{q}}$ are written to a csv-file at every time step. One csv-file is generated per test case and these are

parsed by an R-script that numerically calculates the jerk. The essential part of the R-script is shown in Listing 6.1. We also logged the initial and final task function values at time $t = 0$ and $t = t_f$.

```
rawdata <- read.csv("csv_data/f20t10K0.csv", header=FALSE)
J <- rawdata[rawdata$V2=='J',3:20]
qdot <- rawdata[rawdata$V2=='qdot',3:20]
t <- rawdata[rawdata$V2=='J',c("V1")]

dt <- t[2:length(t)] - t[1:length(t)-1]
dt <- c(dt, sum(dt)/length(dt))

t_red <- t[2:(length(t)-1)]
dt_red <- dt[2:(length(dt)-1)]

de <- rowSums(J*qdot)
de <- unname(de)
dde <- (-0.5*de[3:length(de)] +0.5*de[1:(length(de)-2)]) / dt_
    red
ddde <- (de[3:length(de)] -2*de[2:(length(de)-1)] +de[1:(
    length(de)-2)]) / dt_red
```

Listing 6.1: R-script for numerical calculation of task jerk.

6.3.3 Test Results

We could see that, as the frequency of the controller was increased, the readings got more unstable. Reasons for this can for example be that the writing of results to a file can affect the CPU power that the controller is given. Another way of measuring this however is by lowering the frequency of the controller and increasing the duration of the motion. That results in the same amount of samplings but in the latter case the effects of high frequencies on the readings are not as prominent. We therefore refer to the number of samples, rather than the controller frequency or the motion duration, when discussing the results in this section.

The desired final task value for this setup was 0, i.e. the two points fixed to the grippers were to be on top of each other. However, this was never fully achieved as can be seen on the left-hand side in Figure 6.4 and Figure 6.5. For this particular task the final task value never gets lower than 1.75% of the initial task value. Reasons for this can be that the implementation of minimal jerk task dynamics is not perfect due to that it is sampling-based and that the kinematic controls are never exactly achieved. We could observe a minimum in the error rate of final task values between 25-30 samplings throughout the motion duration, see Figure 6.4, and for values of K higher than 1, see Figure 6.5.

When looking at how well the minimal jerk property of the generated motion was achieved we could see that the jerk increased with the number of samplings and with increasing K. We believe that the low rations found at low durations, see the right-hand side of Figure 6.4, are due to that at these few samplings the controller never reaches the desired jerk which then results in the final error rate being high. As the jerk is numerically derived from the task velocity we see that by increasing the number of samplings would also increase the motion jerk. The more changes there are in the desired task velocity, the more total jerk. This can be seen as the duration increases in Figure 6.4.

The Σ/C ratio[5] increases drastically with increasing K, see the right-hand side of Figure 6.5. The plateau in the graph is partially due to that the values on the x-axis are distributed logarithmically (which enlarges the plateau visually). The steady increase of Σ/C is however also seen in Figure 6.6a and 6.6b. When K is increased, the controller is more and more likely to react on deviations from the perfect minimal jerk trajectory. This results in increased jagged behaviour of the motion acceleration and thus increases the total jerk.

From all these observations we can see that for this particular task a good number of samplings lies between 18-25 throughout the motion, and a K-value between 1-5. That would imply a final error rate at under 2% and 5% increase in total jerk compared with the perfect analytic minimal jerk trajectory.

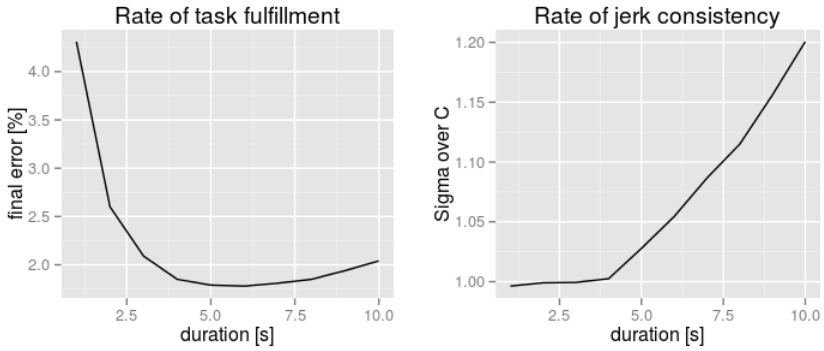


Figure 6.4: Final task error and total jerk ratio as functions of motion duration. The controller frequency was 5 Hz.

[5] C is the analytic cost value found in Equation (6.10). Σ is the numerical version of this, i.e. the sum of `ddde` in Listing 6.1.

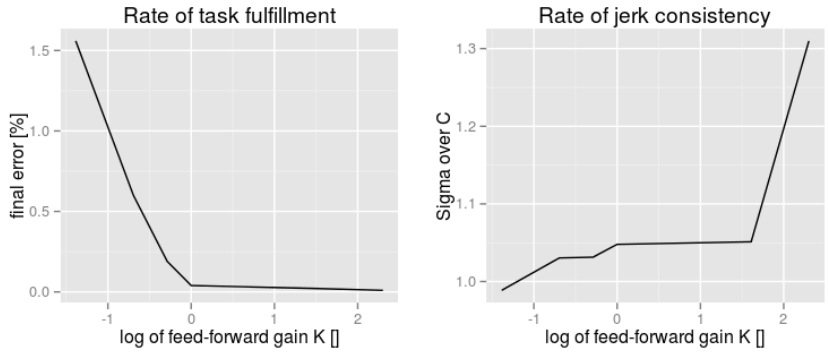
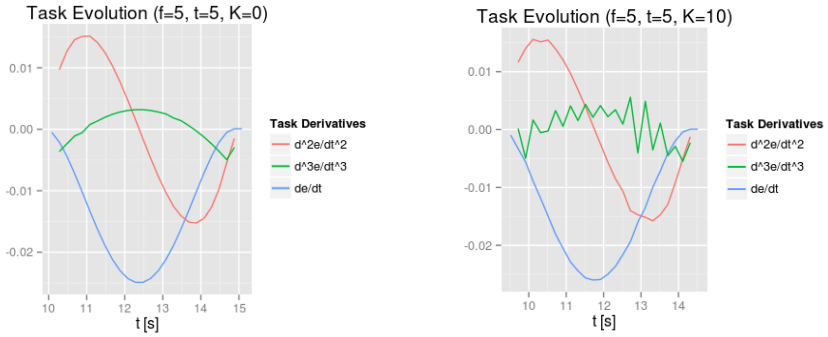


Figure 6.5: Final task error and total jerk ratio as functions of K. `log` refers to the natural logarithm. The controller frequency was 5 Hz.



(a) Derivatives of the task function for $K = 0$ at $5Hz$ during $5s$. The green line shows smooth motion jerk.

(b) Derivatives of the task function for $K = 10$ at $5Hz$ during $5s$. The green line shows jagged motion jerk.

Figure 6.6: Two graphs showing task velocity evolution with acceleration and jerk computed using finite difference coefficients.

Table 6.8: Results from minimal jerk evaluation tests.

$f\ [Hz]$	$t_f\ [s]$	$K\ []$	$e(t_0)\ [mm]$	$e(t_f)\ [mm]$	$\frac{e(t_f)}{e(t_0)}\ [\%]$	$\Sigma/C\ []$
5	5	0	66.8478	2.776140	4.15	0.996328
7	5	0	66.7390	2.207200	3.31	0.9986536
10	5	0	66.8675	1.764370	2.64	1.001305
20	5	0	66.8341	1.193240	1.79	1.027811
35	5	0	66.8068	0.930022	1.39	1.074604
70	5	0	66.8431	0.738589	1.10	1.167346
140	5	0	66.7979	0.642059	0.96	1.325218
280	5	0	66.7485	0.592049	0.89	N/A
550	5	0	66.8262	0.557443	0.83	N/A
1000	5	0	66.7537	0.546637	0.82	N/A
20	1	0	66.8181	2.88184	4.31	0.9963557
20	2	0	67.0790	1.74175	2.60	0.9990655
20	3	0	66.7654	1.39452	2.09	0.9994530
20	4	0	66.9984	1.24228	1.85	1.0025070
20	5	0	66.8946	1.18913	1.78	1.0267550
20	6	0	66.8160	1.18773	1.78	1.0544740
20	7	0	66.8283	1.20804	1.81	1.0866080
20	8	0	66.9008	1.24011	1.85	1.1149260
20	9	0	66.8777	1.30059	1.94	1.1560850
20	10	0	66.8497	1.36504	2.04	1.200642

Table 6.9: Results of closed-loop minimal jerk dynamics (with added P-controller) applied on a 1d-point-on-point projection task.

$f\ [Hz]$	$t_f\ [s]$	$K\ []$	$e(t_0)\ [mm]$	$e(t_f)\ [mm]$	$\frac{e(t_f)}{e(t_0)}\ [\%]$	$\Sigma/C\ []$
5	5	0.25	66.8145	1.04379	1.56	0.9886755
5	5	0.5	66.7367	0.400301	0.60	1.030462
5	5	0.75	66.8091	0.127858	0.19	1.031439
5	5	1	66.8207	0.0296274	0.04	1.047914
5	5	5	66.7469	0.0154645	0.02	1.051242
5	5	10	66.8388	0.00773114	0.01	1.31

Chapter 7

Conclusions

In this thesis we have investigated how a local optimization based motion planning and generation framework can be implemented in software. We wanted a modular framework that could automatically translate task and task space definitions customized by an end user to quadratic programs that were ordered and solved hierarchically. To do this, we set up software quality attributes and designed a framework that was implemented in C++. The framework is independent of which third-party optimization solver library is used, and also independent of which third-party communications systems library is used to communicate with a robot. Our implementation also provides the use of ROS as a communications system, and Gurobi as an optimization solver through the CasADi library. In the oncoming subsections we discuss the results of the evaluations made.

Software Quality

We find that whether or not the quality attributes setup for this project were satisfactorily fulfilled was not determinable from the evaluations made of the source code. Partially, this is due to that our research question on software quality was not posed to give a quantitative measure. However, due to the nature of these quality attributes it is very hard to extract quantitative measures that are fully comparable between software projects. The Microsoft Foundation Classes versions that we compared our framework with, v. 1.0-5.0, were developed in 1992-1997 [2]. With the increased knowledge created in software quality, and techniques such as design patterns, the results showing that HiQP has similar software quality levels as MFC did during its development in 1992-1997 we find reasonable and sound.

To address the two areas where HiQP seems to fall short compared to MFC, id. est. understandability and effectivity, we conclude the following. Understandability is negatively impacted by the measures of polymorphism and coupling. While polymorphism affects four quality attributes positively and only one negatively we instead propose changes to be made that decreases coupling in the code. High coupling in the current version of HiQP is primarily found in classes that use the geometric primitive classes. Currently, in these cases all geometric primitive types must be accounted for at every such occasion therefore the coupling gets high. Finding another design that addresses this problem is crucial to reduce coupling in HiQP. Regarding effectivity, we noted that abstraction is higher for MFC than it is for HiQP. Abstraction, when measured as the average number of ancestors, in HiQP can be difficult to increase since all hierarchies have only one level and inherit in tree-structures. Thus, classes in HiQP have at most one ancestor currently. One way of changing the code to increase abstraction could be to make a task interface that inherits from both `TaskFunction` and `TaskDynamics`. Also, among the geometric primitives a lot of the data representations available in the point primitive and in the line primitive are relevant to the higher order primitives such as cylinder and sphere. Here changing the inheritance structure to let higher order primitives inherit from lower order ones could lead to a higher level of abstraction.

Framework Functionality

Since only 55% of the (globally feasible) test cases for gripping a cylinder were successful, we conclude that when using the HiQP framework in a gripping context with a 7-DoF revolution-joint robot arm one has to be careful of the setup of the grabbing tasks. More generally, using such a local optimization approach to motion generation requires an analysis of how tasks are setup predicated on certain initial configurations. More specifically, in a one task only case, if a goal position in task space is visible from the initial position it seems likely that the HiQP controller will generate a valid motion. Also, setting up tasks that are out of reach can cause problems if a set of different behaviours are to be executed in series using for instance a finite state machine. If the finite state machine is triggered on the successful achievement of tasks the process of behaviour execution can be brought to a halt[1]. Because of this the benefits of using a local optimization approach such as

- being able to compute controls online at runtime,

[1]Such failure behaviour can be detected in task progress stagnation, but the FSM might still be unable to recover from stagnation.

- being able to pose constraints,
- being able to react to disturbances or changes in the environment,
- being able to exploit redundancy,
- being able to use feedback control for more precise task execution,
- begin able to account for multiple tasks simultaneously,
- being able to enforce a strict hierarchy of tasks

come at the cost of careful implementation and execution of tasks. Addressing the issue of incompleteness of this approach therefore becomes important to extend the usage of tasks to generate globally feasible solutions for motion generation. We discuss this further in Section 7.1.

Minimal Jerk Task Dynamics

The evaluation of our minimal jerk implementation shows that it is possible to generate minimal jerk motions for simple tasks using local control optimization. Since the gain K of the feed-forward part of the controller is reacting on deviations from the minimal jerk trajectory, a set of undesired outcomes can be identified.

- If the position of the task is pushed away from its trajectory by a higher priority task, the feed-forward controller will react stronger the further away from the trajectory the task function value is. This will not override the constraints set by higher priority tasks as lower priority tasks will be solved for in the null-space of the higher prioritized ones. However, the feed-forward might produce rapid changes in acceleration inside that null-space that will result in higher motion jerk throughout the task evolution which might be undesirable.
- During task evolution the computed optimal controls can be executed more or less effectively by the underlying velocity controller. When the effectivity of executing these controls vary during motion execution, a static K-value will have different impacts. At times when the task function value is deviating more from its reference value a certain level of feed-forward control will result in a more jerky behaviour, which is not desirably when using minimal jerk dynamics.

Thus, setting K to values larger than 0 to utilize the feed-forward controller requires careful planning of how the tasks are setup. This can improve both the fulfillment of the task as well as the overall jerk minimization,

but can also lead to undesired behaviour. We note also that rapid changes of the task dynamics due to K will lead to rapid changes of the null-space left over to lower priority tasks. Therefore, their performance can possibly be undesirably affected by this feature. To gain more insight into this, one would have to evaluate the use of feed-forward control when running multiple tasks in parallel at various priority levels.

Relation to Global Path Planning

The main issue with a local optimal control approach as with HiQP, which is greedy in a temporal sense, is that tasks can stagnate before having reached their goal state. To battle this issue one could use an internal model representation of the robot and its environment to try to optimize controls into the future[2]. However, using sampling-based path planning solvers one is often able to find a global solution, but perhaps not an optimal one, in much shorter time than when using a fully optimal control that is optimal also with respect to time. Graph based path planning approaches allow to find an optimal solution at a certain discretization level but the time consumption grows exponentially with the number of degrees of freedom. Using control also benefits task execution accuracy as it involves feedback at run-time. Planning based controllers need to re-plan upon changes in the environment. Also, they need to re-plan upon task fulfillment in order to make up for any deviations from the goal state.

7.1 Future Work

Currently HiQP comes with a velocity controller implementation for ROS. As many robots do not have hardware interfaces supporting velocity controls implementing a position controller and an effort controller would be beneficiary. Implementing a position controller would be quite straight forward as one would only have to use the current joint positions and the velocity controls to produce desired joint positions. Implementing an effort controller would involve implementing a general way of representing an internal model of the robot. This could be done for example by having an interface parameterized using types from Orocos KDL. It would then be up to the user to implement a class inheriting from this interface that performs internal simulation of the robot in real time.

Having such an internal robot model interface also allows for predicting the future state of the robot given the generated controls. As in model predictive control (MPC) the optimization can then be carried out over a

[2]Known as Model Predictive Control.

time span instead of only instantaneously. Since the task jacobian would still be instantaneously constant for each time step, but not the same between time steps, there is reason to belief that modern hardware would be able to run optimization and future prediction inside some time horizon, effectively this would extend into an MPC controller. Another benefit of predicting the future could be to try to avoid getting stuck inside local minima. If the time horizon given to the optimizer is long enough to find more than one local minima, these can be stored and used when the task function reaches a minima to try to get out of the locally optimal region.

Bibliography

[1] ABB. *YuMi*. `http://new.abb.com/products/robotics/industrial-robots/yumi`. [Online; accessed 5-october-2016]. 2015.

[2] J. Bansiya and C. G. Davis. "A hierarchical model for object-oriented design quality assessment". In: *IEEE Transactions on Software Engineering* 28.1 (2002), pp. 4–17.

[3] Jerome Barraquand and Jean-Claude Latombe. "A Monte-Carlo algorithm for path planning with many degrees of freedom". English. In: 1990, pp. 1712–1717.

[4] Jérôme Barraquand and Jean-Claude Latombe. "Robot Motion Planning: A Distributed Approach". In: *The International Journal of RObotics Research* 10 (1991), pp. 628–649.

[5] Dmitry Berenson, Siddhartha Srinivasa, and James Kuffner. "Task Space Regions: A Framework for Pose-Constrained Manipulation Planning". In: *International Journal of Robotics Research (IJRR)* 30.12 (Oct. 2011), pp. 1435–1460.

[6] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge, United Kingdom: Cambridge University Press, 2004.

[7] L. C. Briand et al. "Exploring the relationships between design measures and software quality in object-oriented systems". In: *Journal of Systems and Software* 51.3 (2000), pp. 245–273.

[8] Patrick Doherty et al. "The WITAS unmanned aerial vehicle project". In: ed. by Werner Horn. Proceedings of the 14th European Conference on Artificial Intelligence (ECAI). IOS Press, 2000, pp. 747–755. ISBN: 1-58603-013-2, 4-274-90388-5.

[9] T. Flash and N. Hogan. "The coordination of arm movements: An experimentally confirmed mathematical model". In: *Journal of Neuroscience* 5.7 (1985), pp. 1688–1703.

[10] B. Fornberg. "Generation of finite difference formulas on arbitrarily spaced grids". English. In: *Mathematics of Computation* 51.184 (1988), pp. 699–706.

[11] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[12] A. Herzog et al. "Balancing experiments on a torque-controlled humanoid with hierarchical inverse dynamics". In: *IEEE International Conference on Intelligent Robots and Systems*. 2014, pp. 981–988.

[13] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. "Movement imitation with nonlinear dynamical systems in humanoid robots". In: *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*. Vol. 2. IEEE. 2002, pp. 1398–1403.

[14] O. Kanoun, F. Lamiraux, and P. Wieber. "Kinematic control of redundant manipulators: Generalizing the task-priority framework to inequality task". In: *IEEE Transactions on Robotics* 27.4 (2011), pp. 785–792.

[15] Sertac Karaman and Emilio Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: *International Journal of Robotics Research* 30.7 (2011), pp. 846–894.

[16] L. E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". English. In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[17] O. Khatib. "A Unified Approach for Motion and Force Control of Robot Manipulators: The Operational Space Formulation". In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53.

[18] Oussama Khatib. "The potential field approach and operational space formulation in robot control". In: *Adaptive and Learning Systems: Theory and Applications*. 1986, pp. 367–377.

[19] R. Krug et al. "The Next Step in Robot Commissioning: Autonomous Picking and Palletizing". In: *IEEE Robotics and Automation Letters* 1.1 (2016), pp. 546–553.

[20] James J. Kuffner Jr. and Steven M. La Valle. "RRT-connect: an efficient approach to single-query path planning". English. In: *Proceedings - IEEE International Conference on Robotics and Automation*. Vol. 2. 2000, pp. 995–1001.

[21] S. M. LaValle and M. S. Branicky. *On the relationship between classical grid search and probabilistic roadmaps*. English. Vol. 7 STAR. Springer Tracts in Advanced Robotics. 2004, pp. 59–75.

[22] T. Lozano-Pérez. "Spatial Planning: A Configuration Space Approach". English. In: *IEEE Transactions on Computers* C-32.2 (1983), pp. 108–120.

[23] LaValle S. M. *Planning Algorithms*. Cambridge University Press, 2006.

[24] Nicolas Mansard et al. "A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks". In: *Advanced Robotics, 2009. ICAR 2009. International Conference on*. IEEE. 2009, pp. 1–6.

[25] Robert C Martin. "Design principles and Design Patterns". In: *Object Mentor* 1 (2000), p. 34.

[26] J. Nocedal and S. J. Wright. *Numerical Optimization*. 2nd ed. 2006, pp. 448–496.

[27] Bashar Nuseibeh and Steve Easterbrook. "Requirements Engineering: a Roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM Press, 2000, pp. 35–46.

[28] L. Saab et al. "Dynamic whole-body motion generation under rigid contacts and other unilateral constraints". In: *IEEE Transactions on Robotics* 29.2 (2013), pp. 346–362.

[29] C. Samson, M. Le Borgne, and B. Espiau. *Robot Control: the Task Function Approach*. Oxford, United Kingdom: Clarendon Press, 1991.

[30] I. Sommerville. *Software Systems Architecture*. 9th ed. Pearson Education, Inc., 2011.

[31] I. A. Şucan, M. Moll, and L. Kavraki. "The open motion planning library". In: *IEEE Robotics and Automation Magazine* 19.4 (2012), pp. 72–82.

[32] V. M. Zatsiorsky. *Kinematics of Human Motion*. 1998.

[33] David Zho and Jean-Claude Latombe. "New Heuristic Algorithms for Efficient Hierarchical Path Planning". In: *IEEE Transactions on Robotics and Automation* 7.1 (1991), pp. 9–20.