Online Collision Avoidance

Akhil Jain, Dominik Sixt & Julian Exner

November 28, 2016
Version: 1.0

University of Bielefeld

Faculty of Technology

Neuroinformatics Group

Report

Online Collision Avoidance

Akhil Jain, Dominik Sixt & Julian Exner

Supervisors Dr. rer. nat. Robert Haschke
Dr. Guillaume Walck

November 28, 2016

Contents

1 Introduction

The next years the usage of robots in industry will be increased so that around 1.3 million industrial robots will be entering service in factories around the world by 2018 [@Int16]. Therefore, it is inevitable that the interaction between robots and workers will raise likewise. The regular contact concludes preventive measures especially for humans, who are working as maintainers or in the same working area and have to interact with robots. The amount of industrial accidents of robots in Germany over the last ten years is nearly similar [@Pat16][@Deu16]. Consequently, the requirement of safety precautions is still current and has to raise.
Beside this issue also the robots have to be secured against damage caused by artificial, non-static objects whose unpredictable motion trajectories are within or cross the robots working area. We will explore a way to avoid collision of manipulators in real-time.

1.1 Focus

This work will focus on the online collision avoidance of a high degrees of freedom manipulator. The developed module is preceding the given trajectory generator to fulfill the work orders. Detection and recognition of the obstacles are off-focus and will be simulated.
For this purpose, first there is an calculation to get generated forces of obstacles to the manipulator. In a further step these forces are processed and applied to the actual working order.

1.2 Structure

First we will focus on the basics of used methodologies and the exploited tools. Afterwards a description of the structure and implementation of the module is given, followed by experiments. In the end we will summarize our results and describe potential future work.

2 Basics

Initially we introduce in the theoretical basics which are used in the development of the online collision avoidance module, followed by a short summary of used frameworks and libraries.

2.1 Artificial Potential Fields

In „Real-Time Obstacle Avoidance for Manipulators and Mobile Robots“, Khatib describes the potential field approach in the context of manipulator as follows:

> *The manipulator moves in a field of forces. The position to be reached is an attractive pole far the end effector and obstacles are repulsive surfaces for the manipulator parts.* [Kha86, section 3]

These forces are derived from an artificial potential field. An artificial potential field U_{art} considering a single obstacle O can be formally described by

$$U_{art}(x) = U_{x_d}(x) + U_O(x), \tag{2.1}$$

where $U_{x_d}(x)$ is the potential generated by the goal position x_d, and $U_O(x)$ the potential generated by the obstacle O. The attractive potential U_{x_d} is described by

$$U_{x_d}(x) = \frac{1}{2} k_p (x - x_d)^2, \tag{2.2}$$

where k_p is a gain factor. The repulsive potential U_O generated by obstacles looks similar:

$$U_O(x) = \begin{cases} \frac{1}{2}\eta \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho \leqslant \rho_0 \\ 0 & \text{if } \rho > \rho_0 \end{cases}, \tag{2.3}$$

where ρ is the minimum distance to the obstacle O and ρ_0 the limiting distance of the potential field. If the minimum distance is greater than this threshold, the generated potential is 0. Below this threshold the potential is positive and grows to infinity when approaching $\rho = 0$. This potential is scaled by η, a linear gain factor.

These potential fields generate the following forces:

$$F^*_{x_d} = -grad[U_{x_d}(x)] \tag{2.4}$$
$$F^*_O = -grad[U_O(x)] \tag{2.5}$$

$F^*_{x_d}$ is the attractive force and F^*_O represents the force generated by the goal and the obstacle, respectively. The total force is calculated by adding these two forces.

$$F^*_{art} = F^*_{x_d} + F^*_O \tag{2.6}$$

The end effector reaches the goal by following the generated force. This corresponds to a gradient descent in the potential field. The disadvantage of this approach is that the end effector may stuck in a local minimum. [Kha86]

2.2 Inverse Kinematics

Receiving requested alterations in the current position of the manipulator we have to calculate the change in each joint. Starting with the equation to solve the forwarding kinematics (2.7) we transpose the equation so we obtain the inverse kinematics (2.8) [Möl15].

$$\Delta\vec{x} = J\Delta\vec{\theta} \tag{2.7}$$
$$\Delta\vec{\theta} = J^{-1}\Delta\vec{x} \tag{2.8}$$

In case the jacobian is non-quadratic, the inversion of this matrix is not feasible, so you have to use the pseudoinverse of the jacobian (2.9) [HS15].

$$J^{\#} = \begin{cases} J^t\left(JJ^t\right) & \text{if rows of J linear independent} \\ \left(J^tJ\right)^{-1}J^t & \text{if columns of J linear independent} \\ \lim\limits_{\delta\to 0} J^t\left(JJ^t + \delta\mathbb{1}\right)^{-1} & \text{otherwise} \end{cases} \tag{2.9}$$

Another option to prevent singularities by inversing the jacobian is to use the Jacobian-Transpose Method (2.10). This equation results in a imprecise solution, though it will almost point in the same direction as Δx. Furthermore, the transposed jacobian also projects wrenches at the end-effector of a manipulator at the joints torques (2.11) [HS15]. Finally we obtain a equation to calculate expected shift using the transposed jacobian and the wrenches (2.12).

$$\Delta\vec{\theta} = \alpha J^t\Delta\vec{x} \tag{2.10}$$
$$\vec{\tau} = \alpha J^t\vec{\omega} \tag{2.11}$$
$$\Delta\vec{\theta} = \alpha J^t\vec{\omega} \tag{2.12}$$

2.3 Used Tools

2.3.1 Robot Operating System

The Robot Operating System (ROS) is a framework giving a collection of tools, libraries and conventions for writing robot software. It aims to produce robust and complex behaviors at several robot platforms.
One of the core components of ROS is the usage as a middleware to connect the realizably count of different components robots need to exercise, e.g. the behavior software connects to sensors and actuators. Therefore the ROS middleware realizes the publish-subscribe pattern, where single components are constructed as nodes which can communicate among themselves via predefined messages. ROS also allows to record these messages, so you can reduce your development effort. Additionally ROS provides powerful tools, which supports introspecting, debugging, plotting and visualizing the robots state. [@Ros]

2.3.2 FCL

For the purpose of continuous distance computation, we used Flexible collision library (FCL). FCL is a C++ library which provides unified interface to perform different queries on a wide range of models. The disadvantages of many libraries proposed before was that they had quite a few queries and they worked on certain restricted number of models, these challenges are overcome by FCL which gives it an upper hand over many such libraries.
FCL handles object by organizing them in the form of meshes and basic geometric shapes which are represented by Bounding volume hierarchy. The complete information required for executing the query is stored in traversal node and then we traverse the hierarchical structure to perform a distance computation. [@Fcl]

2.3.3 Orocos

To empower the system with real time capability and to ensure the data exchange between different components of our system, we used a C++ based framework called Open Robot Control Software (Orocos). The Orocos project supports four C++ libraries: the Real-Time Toolkit, the Kinematics and Dynamics Library, the Bayesian Filtering Library and the Orocos Component Library. KDL was used in our project to represent kinematic structures and compute inverse kinematics thus providing high level management of interaction within the application.

Also, the existing trajectory generator used Orocos which necessitated us to use it for writing our component. [@Oro]

2.4 Environment

The Neuroinformatics Group of the University of Bielefeld convenience a grasp labor with several robot platforms specialized in manipulators. In this project a seven degrees of freedom manipulator manufactured by KUKA (see figure 2.1) is provided to us. For testing purposes this environment is also simulated, so modules can be tested coincidentally and the setup is protected against permanent damage. For this purpose tools given by ROS (see section 2.3.1) and Gazebo [@Gaz], a powerful robot simulation can be used.

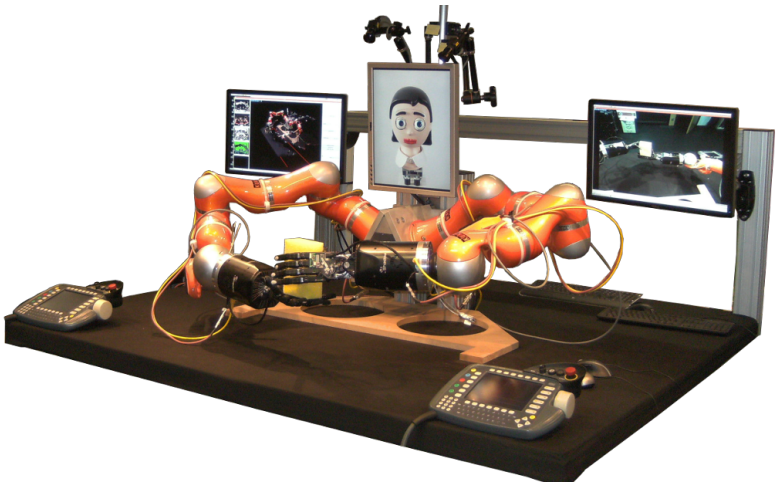


Fig. 2.1.: Robot platform FAMULA in the CIT-EC, University of Bielefeld, including the KUKA LWR 4 with 7 DOF [Wal16]

3 Implementation

3.1 Overview

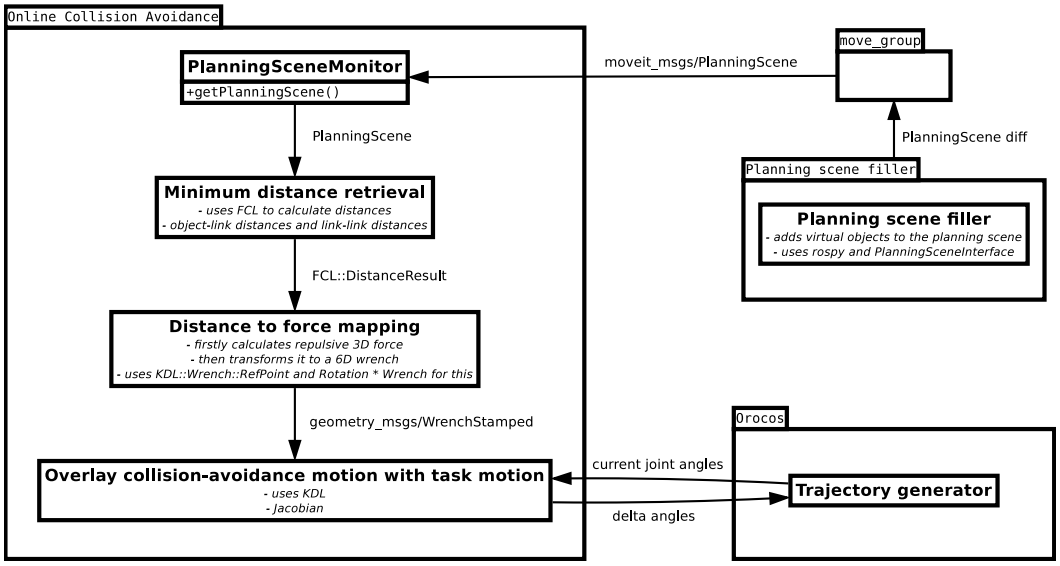


Fig. 3.1.: Overview of the implementation

The main implementation consists of a single ROS node called *Online Collision Avoidance*. This node contains the minimum distance retrieval between the robot's links and obstacles, mapping those distances to forces and calculating joint angle deltas from those forces. The main input to this node is the current planning scene, an inner representation of the robots working area, maintained by MoveIt!'s *move_group* node. By monitoring the planning scene, the online collision avoidance node is notified by changes in the environment. The planning scene may be modified by an object recognition node. For testing purposes the *planning scene filler* node was implemented using Python. This node creates virtual objects and adds them to the planning scene.

The *Online Collision Avoidance* node runs in an endless loop waiting for updates in the planning scene until a shutdown is requested. Whenever the planning scene changes, new distances, forces and joint angle deltas must be computed. Theses

resulting angle deltas are communicated via a ROS topic to the lower level hardware components. Here they are combined with the initial trajectory's joint angles.

3.2 Minimum Distance Calculation

First task was to insert and identify the known objects in planning scene environment. We begin with initializing the ROS node and accessing the planning scene using the planning scene interface. Then, after we prepare the pose box and add the object to the scene, the scene is published. We continue updating the object's position until the node is shut down. The rate at which we update the planning scene is defined by ROS's rate function which executes the loop at the desired frequency.

Algorithm 1: Planning scene filler

Result: PlanningScene filled with virtual objects
initialization
while *true* **do**
    get all virtual objects
    for *all objects* **do**
        transform coordinates into planning scene coordinates
        add object to planning scene
    end
    publish planning scene (diff)
end

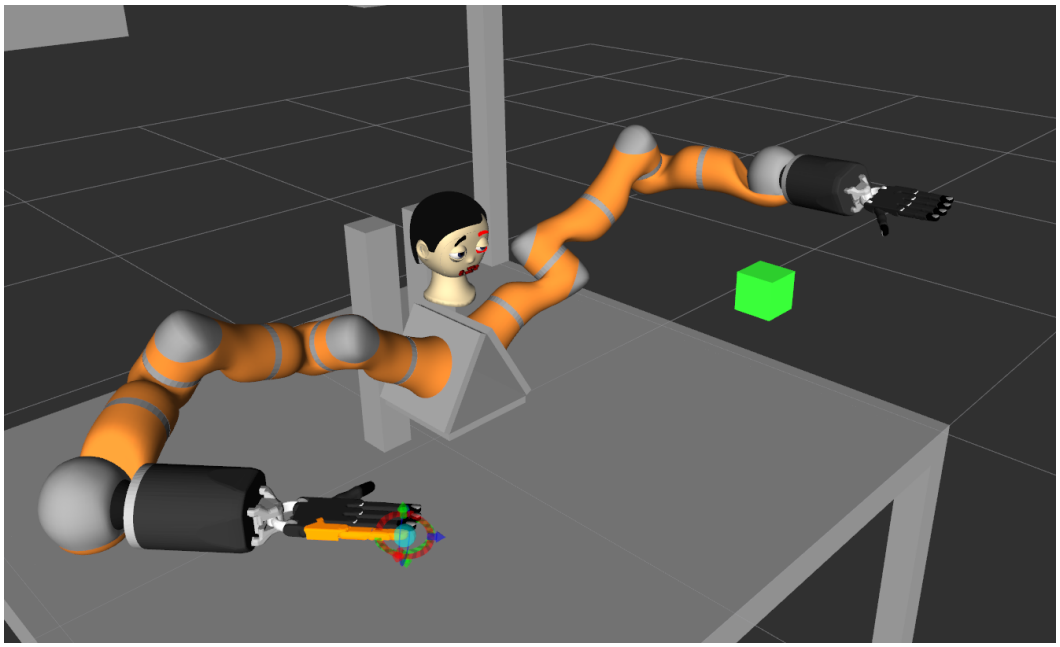


Fig. 3.2.: Planning scene filled with the box (in green)

After getting the planning scene objects, next step was to find the distance from robot link to the object and from link to link (to avoid self collision). For computing

the distance, FCL library was used as it performs distance query using geometric models composed of triangles. We set the geometry of the objects as mesh soup and transform of the objects as rotation and translation matrix, combining them together we get collision request data structure. Both the representation of the world and the kinematic model for which the planning scene is maintained was accessed using the planning scene.

Since our aim was to plan the trajectory, we applied force only to the arm links. The generated force on the arm depends on its distance from collision objects as well as its distance from other links. In our case we only considered force generating from the other arm and the frame as the distance of a link from other link in the same arm is negligible. Also we don't generate the force for the lowest pair of the right and left arm as they can't avoid each other and are fairly close, so they generate a big force on one another. Another problem we faced while dealing with frame was that many object has the same id, namely "frame". As such we observed the fluctuation when we generated the force because the objects having same id had different distances. To tackle this problem we added the counter in id so that it has different id every time.

Once the objects are set, we perform the distance computation between them. We set the query request data structure and then run the query function by using request as the input and result which has query result data structure as the output. Also, we transform it into world coordinate which is required when one of the object is a geometry primitive/shape. It seems to be a bug in FCL, check out for more information `https://github.com/flexible-collision-library/fcl/issues/127`. Finally, the output we get is the distance of links from other links and links from other collision objects and we filtered out the links we don't need for our trajectory generation. The calculated distances are then visualized using RViz, a 3D visualizer of the ROS framework (see figures 3.3 and 3.4).

Algorithm 2: Minimum distance computation

Result: Distance between collision objects and links
wait for planning scene
find all obstacle and robot objects
get all the link arm position and set the list of link in which force is applied and links that generates the force
for *each link* **do**
    get geometry data and link name
    for *each link* **do**
        get the geometry data and link name
        filter out the links we dont want to consider
        find minimum distance between two links. if two objects are in collision, min_distance $\leqslant 0$
        return distance
    end
    for *each object from planning scene* **do**
        get the collision object
        find minimum distance between the link and object. if two objects are in collision, min_distance $\leqslant 0$
        return distance
    end
end

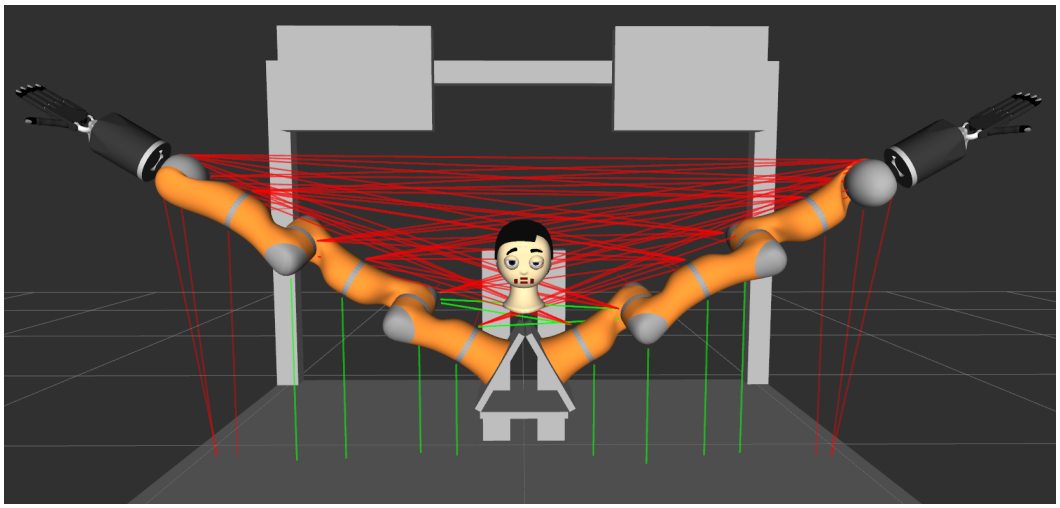


Fig. 3.3.: Link to link distance: green implies a distance less than 0.6 meter and red stands for distances more than 0.6 meter.

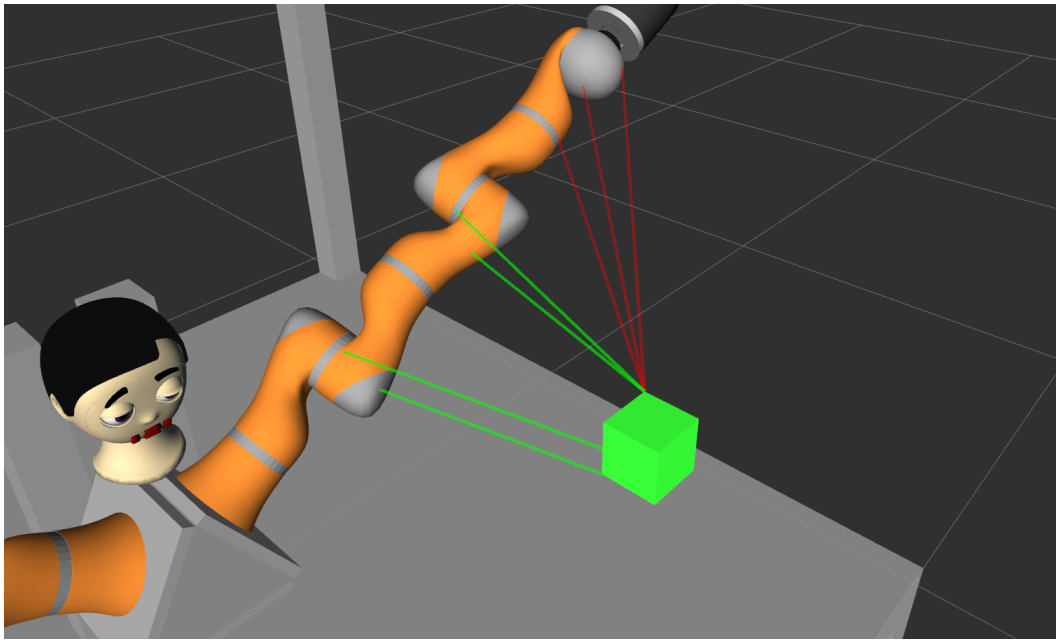


Fig. 3.4.: Link to obstacle distance: green implies a distance less than 0.6 meter and red stands for distances more than 0.6 meter.

3.3 Distance to force mapping

The minimum distances between links and links, and links and obstacles must be mapped to forces, or more precisely wrenches, to be applicable to the robot's joint state. The pseudo code 3 presents the overall outline of the approach. The individual steps are explained in more detail below.

Algorithm 3: Mapping distances to forces

Input: A tuple that contains information about the two objects and the distance between them.

Output: A force.

wait for new ⟨⟨obj, obj⟩, dist⟩ tuple
`// calculate repulsive potential:`
if $dist <= max\_distance$ **then**
  $F_{rep} = -\eta \left(\frac{1}{dist} - \frac{1}{max\_distance} \right) \left(-\frac{1}{dist^2} \right)$
  calculate direction of force (from object to link)
  scale force by repulsive force
  publish ⟨link, force⟩ to overlay
end

Mapping distances to forces relies on FCL distance results. A FCL distance result contains the minimum distance, the nearest points on both objects and further information about the original objects. If the minimum distance is below a certain threshold, up to which a force should be generated, the magnitude of the force is calculated. Combining equation 2.5 and 2.3 yields

$$F_{rep}(\rho) = -\eta \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right)\left(\frac{1}{-\rho^2}\right) \tag{3.1}$$

Figure 3.5 shows a plot of this function. The direction of the force depends on the location of the nearest points. In the case of an obstacle and a link, the force's direction is equivalent to the one of the vector pointing from the obstacle to the link. This vector is then normalized and scaled by the result of 3.1.

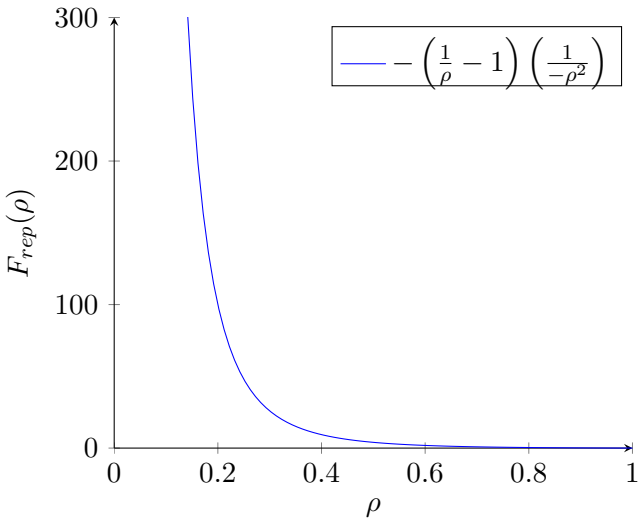


Fig. 3.5.: Plot of $F_{rep}(\rho) = -\eta \left(\frac{1}{\rho} - \frac{1}{\rho_0}\right)\left(\frac{1}{-\rho^2}\right)$ (eq. 3.1) with $\eta = 1$ and $\rho_0 = 1$. $F_{rep}(\rho)$ is 0 for values of $\rho > \rho_0$. For display purposes values of $F_{rep} > 300$ are truncated in the plot.

To apply those three-dimensional forces on the robot's joints, they have to be transformed into six-dimensional wrenches, consisting of a three-dimensional force and a three-dimensional torque. So far, the reference point of the force was the closest point on the link to the obstacle. This is changed to the link's parent joint using KDL. Transforming the force to a new reference point creates a torque. The force stays intact. This now six-dimensional wrench's reference frame is the global frame. But for further calculation it will be changed to the link's frame. These wrenches can then be visualized using RViz. Figure 3.6 shows a visualization of the wrenches.

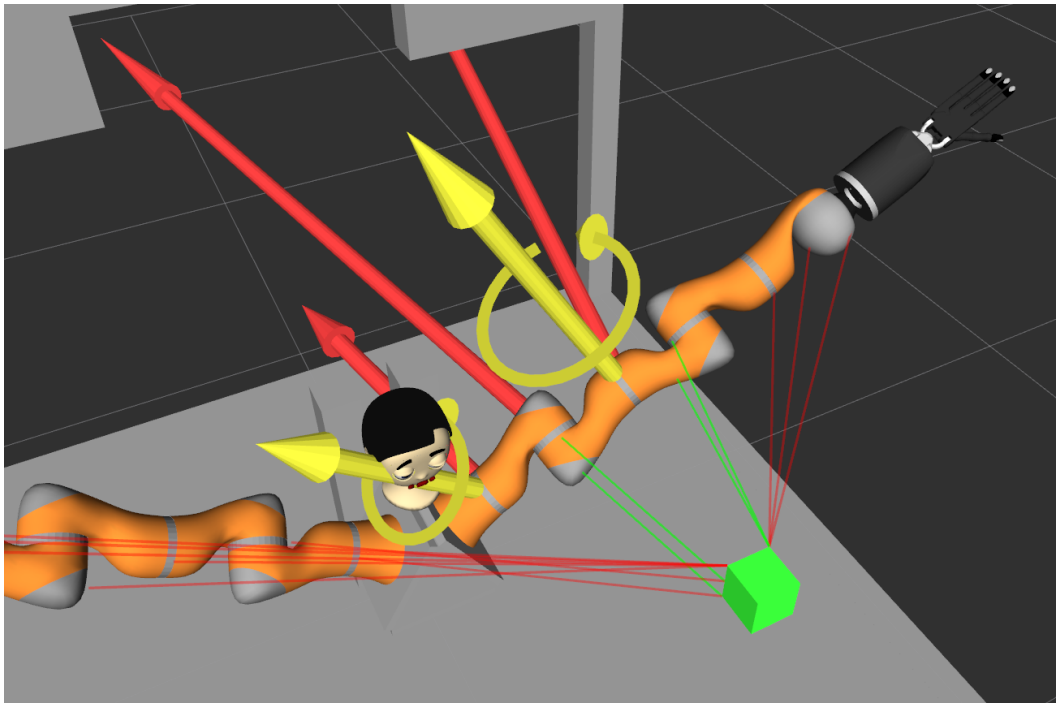


Fig. 3.6.: Visualization of wrenches in RViz: The forces are displayed in red, torques are yellow. The displayed wrenches resulted from considering only forces generated by the green obstacle.

3.4 Overlay collision-avoidance with task motion

To generate the evading motion of the manipulator we have to apply the considered forces described in 3.3 to the actual joint states. The pseudo code 4 represents the functionality of the `OnlineCollisionAvoidanceOverlay::calculateDeltas()`, significant design decisions are described below.

Applying the calculated wrenches as input exploiting the kinematics of a manipulator the actual state of the robot is needed. The Kinematics and Dynamics Library (KDL) provides classes to represent a manipulator as a set of segments, which involves a joint and a frame [@Kdl]. Thus from the robot state a serial `KDL::Chain` of the root to the end-effector is generated.
Involving equations 2.8 and 2.12 we have to calculate the jacobian till each joint, which can be done with the `KDL::ChainJntToJacSolver`. Then the individual joint jacobians are hooked, in case of the 7 joints of the KUKA manipulator the overall jacobian supposed to be 7×42.
Afterwards the $\Delta\theta$ is initiated and includes the wrench of each joint. The $\Delta\theta$ is used as the input vector for the equations 2.8 (with a pseudo inverse of the jacobian $J^{\#} = \left(J^t J\right)^{-1} J^t$, because the columns are linear independent) and 2.12. Also there is the factor α added to interfere and adjust the results. For now the value of α is set

Algorithm 4: Calculation of deltas in joints

Input: Wrenches.
Output: JointState that contains information about the requested adjustment of each joint.

update robot state
get tree of the manipulator
convert tree into chain
initiate overall jacobian
for *all joints* **do**
  calculate jacobian till joint
  add joint jacobian to overall jacobian
end
initiate $\Delta\theta$
for *all wrenches* **do**
  convert wrenches and add to $\Delta\theta$
end
if *jacobian-transpose-method* **then**
  apply jacobian transpose method $\Delta\vec{\theta} = \alpha J^t \vec{\omega}$ to calculate joint deltas
else
  apply pseudo inverse method $\Delta\vec{\theta} = \alpha \left(J^t J\right)^{-1} J^t \Delta\vec{x}$ to calculate joint deltas
end
create message and publish JointState

to zero. Only one method is calculated and can be switch by an boolean while the class is initiated. As result we get the deltas in joints.
At the end all the calculated joint deltas are inserted in a `sensor_msgs::JointState`, which can be send via ROS to the modified trajectory generator described in the next section 3.5.

A problem detected during testing of the system, was that the manipulator oscillated. To prevent this issue the change in the joint deltas is limited. The maximal allowed change in deltas depends on the elapsed time between update steps of the system and the absolute of the maximum joint velocity. If the change in a joint delta is greater than the maximum allowed change, then the new joint delta is truncated to the sum of the old value and maximum allowed change. This applies for a positive change in a joint delta. If the change is negative, it is limited to the difference of the old joint delta and the maximum allowed change.

3.5 Modified Trajectory Generator

The already existing Orocos trajectory generator interpolates between key points of a given trajectory and then publishes the resulting joint positions and velocities to be performed by the hardware. The interpolation is done using splines to ensure that

velocity profiles are smooth. In order to avoid collisions, the performed trajectories need to modified, but not the initially requested one. Otherwise, the requested trajectory will not be reached. It poses as the attracting potential in terms of the potential field approach (see section 2.1). After calculating the angle deltas in joints, as presented in the previous section, they are published on a ROS topic and received on a Orocos port by the trajectory generator. The trajectory generator recalculates the splines and then adds the deltas in joints to the resulting joint positions.

During the testing phase of this project, we encountered that the manipulator oscillated. We suspected high latency, due to the high load of the host system, to be the cause of this problem. This high load was mainly caused by the Gazebo simulation. Therefore, a fake trajectory controller was introduced, which renders the physical simulation carried out by Gazebo redundant. The fake controller does not rely on a physical model and simply linearly interpolates the requested trajectory. The resulting joint states are directly adopted by the robot state. After the linear interpolation, the joint angle deltas are added again.

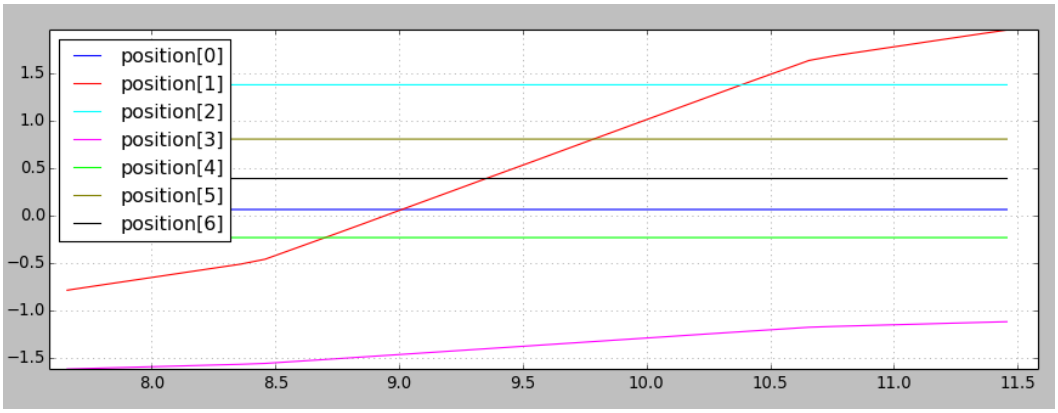
4 Experiment

To test the developed module described in 3.1 as a simulation, there are a static and dynamic testing node provided as simple python scripts.
In the static test node a fixed obstacle can be placed in the working area of the manipulator. Using MoveIt!'s RobotCommander single joints can be set, subsequently a trajectory will be planned and executed, if the goal is reachable. For testing purpose the manipulator starts in a predefined home position (see figure 3.2) and only the joints 1 and 3 are moved, so the trajectory is nearly parallel to the plane and is planned through the obstacle . This precisely control of the manipulator permits reproducible tests and supports the measurement of the systems behavior.
There is also a dynamic test node to analyze the performance. Therefore again a obstacle will be generated and will move from different starting positions, speeds and angles to the active manipulator.
As mentioned in section 2.3.1 the sent messages can be recorded as rosbags and then the behavior of the manipulator can be analyzed.

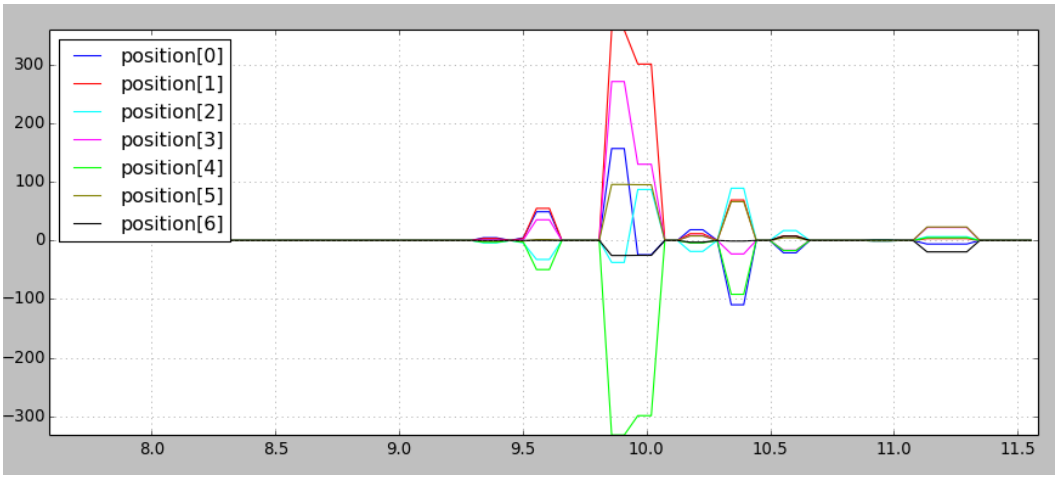
4.1 Results

The graphs in figures 4.1 and 4.2 show the results of the conducted static test mentioned in section 4. In a first test, no limitation of the change in joint angle deltas is applied (see the last paragraph of section 3.4). In figure 4.1a we see the unaltered joint states of the robot's left arm. Joint 1 and joint 3 perform the requested trajectory. Subfigure 4.1b shows the calculated deltas. As we can the see by the axis scaling the values of the determined deltas are much bigger than the initial joint angles. Figure 4.1c shows the actually performed trajectory. It is the sum of the original trajectory (4.1a) and the joint angle deltas (4.1b). The original trajectory becomes insignificant.

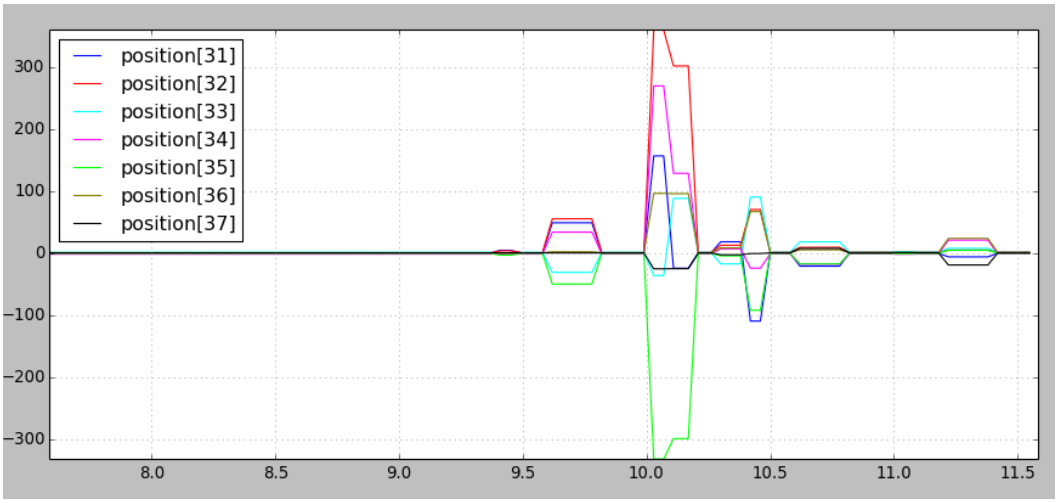
The second test, that includes the limitation of change in joint deltas, is shown in figure 4.2. Compared to angle deltas in 4.1b, the deltas in 4.2b are limited to a more sensible domain. The mixture of the original and the collision avoidance joint states yields a trajectory that is more similar to the requested trajectory. This way the manipulator avoids the obstacle, but still reaches the requested goal.



(a) Original joint states without collision avoidance

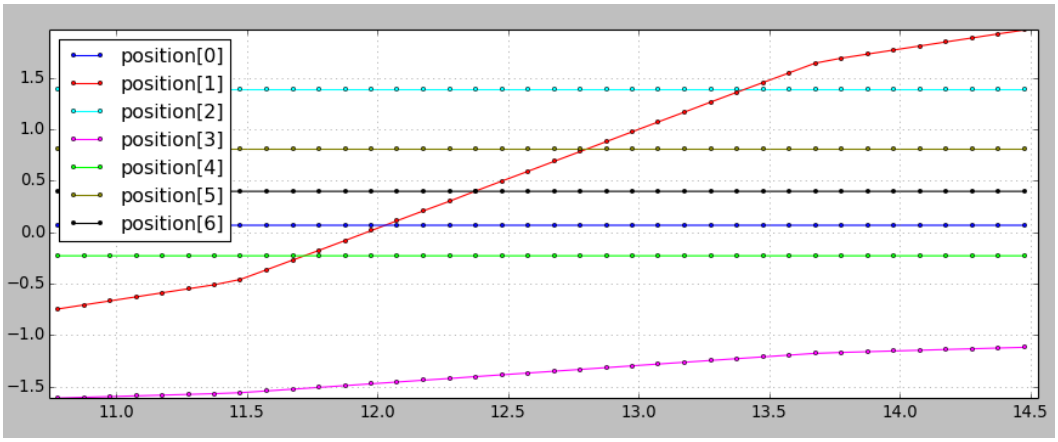


(b) Angle deltas calculated by the collision avoidance

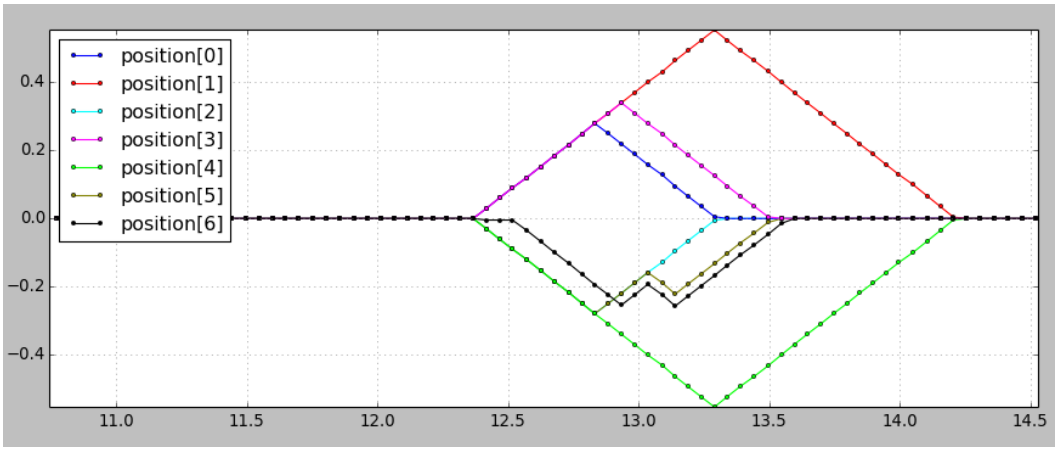


(c) Final joint states including collision avoidance deltas

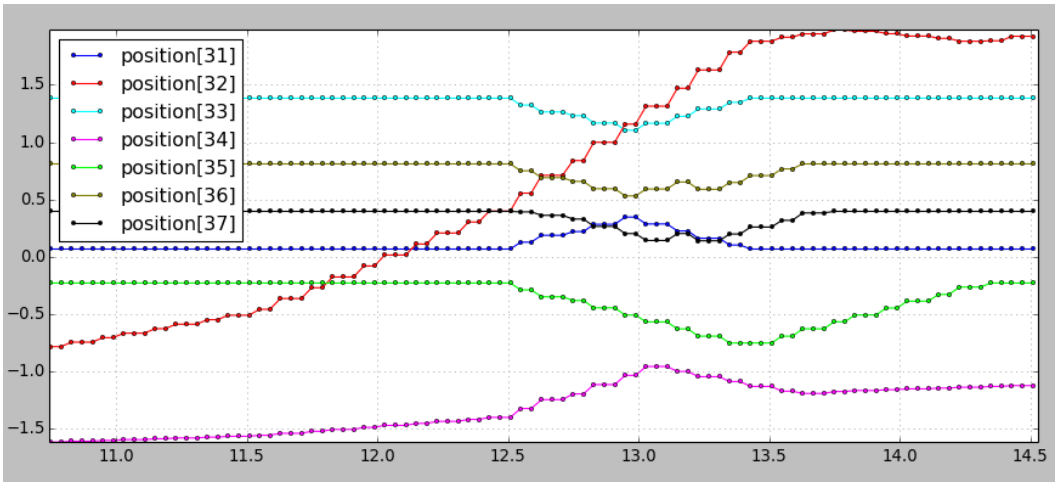
Fig. 4.1.: Above graphs show the behavior of the joint states without a limitation of the collision avoidance angle deltas. The x-axis displays the time. The y-axis shows the values of the joint states in radian.



(a) Original joint states without collision avoidance



(b) Angle deltas calculated by the collision avoidance



(c) Final joint states including collision avoidance deltas

Fig. 4.2.: Above graphs show the behavior of the joint states with the limitation in change of the collision avoidance angle deltas. The x-axis displays the time. The y-axis shows the values of the joint states in radian.

5 Conclusion

In this project, we implemented a collision avoidance strategy. Our tests showed that our implementation is feasible for online collision avoidance. Although, it was implemented for the KUKA LWR 4, it may be adapted to any seven degrees of freedom manipulator. Since oscillation posed a problem in the first testing phase, the system wasn't tested in the real world to avoid damage of the robot.

5.1 Future Work

Further inquiry could be an expansion of the measurement of the system in test cases. For a higher test complexity, there can be generated further obstacles of different size and shapes as well as several obstacles in one test case. Also the working order of the manipulator can be elaborated more complicated. Furthermore we could test the developed module on a real world manipulator to detect unknown influences, which have to be included in the system.
The observation of the accuracy and efficiency of the jacobian-transpose method and the implementation using the pseudo inverse jacobian mentioned in section 2.2 could also be intensified.
Recently the module could also be tested on other manipulators than the KUKA LWR 4. Especially the usage on low degrees of freedom manipulators or in an working environment with more than one manipulator as in figure 2.1 shown could be worthwhile.

Appendix

A

Listing A.1: ROS launch file `oca_test.launch` as used for the tests in section 4.

```
<launch>
  <!-- Parameter for overlay -->
  <!-- gain factor after Jacobian is applied -->
  <param name="alpha" value="1" />
  <!-- suppress output -->
  <param name="print_infos" value="false" />
  <!-- wether joint_deltas are calculated by classic jacobian
      transpose method or by jacobian transpose method using
      pseudoinverse -->
  <param name="jacobian_transpose_method_classic" value="false" />
  <!-- from this velocity and the time delta in the loop, the maximum
       joint delta is computed -->
  <param name="max_joint_velocity" value="0.6" />

  <!-- Factor multiplied with (pi/2) as variable values of joint_1
      -->
  <param name="test_joint_factor" value="1.75" />
  <!-- Height (position.z) of the obstacle in static test -->
  <param name="test_obstacle_height" value="1" />

  <!-- distance up to which forces are generated -->
  <param name="max_distance" value="0.3" />
  <!--  -->
  <param name="force_gain" value="1.0" />
  <!-- ros rate of the main loop -->
  <param name="rate" value="100" />

  <!-- whether forces are generated by the robot's links and table/
      frame -->
  <param name="calculate_link_forces" value="false" />


  <!-- Please source devel/setup.bash -->
  <node name="oca_test" pkg="oca_test" type="oca_test_static.py" />
  <node name="online_collision_avoidance" pkg="
      online_collision_avoidance" type="
      online_collision_avoidance_node" output="screen" />
</launch>
```

Bibliography

[HS15] Robert Haschke and Jochen Steil. *Robotik - Autonomes Greifen*. Universität Bielefeld, 2015, pp. 23,26–28 (cit. on p. 4).

[Kha86] O. Khatib. „Real-Time Obstacle Avoidance for Manipulators and Mobile Robots“. In: *The International Journal of Robotics Research* 5.1 (Mar. 1986), pp. 90–98 (cit. on pp. 3, 4).

[Möl15] Ralf Möller. *Vorlesung Robotermanipulatoren*. Universität Bielefeld, Mar. 2015, 138 pp. (Cit. on p. 4).

[Wal16] Guillaume Walck. *Presentation: Lab Robot Introduction*. Universität Bielefeld, Apr. 2016 (cit. on p. 6).

Websites

[@Fcl] *FCL library*. Nov. 2016. URL: http://wiki.ros.org/fcl (cit. on p. 5).

[@Gaz] *Gazebo Simulation*. Nov. 2016. URL: http://gazebosim.org/ (cit. on p. 6).

[@Kdl] *Orocos Kinematics and Dynamics Library*. Nov. 2016. URL: http://www.orocos.org/kdl (cit. on p. 13).

[@Oro] *Orocos components*. Nov. 2016. URL: http://www.orocos.org/ (cit. on p. 6).

[@Ros] *ROS Core Components*. Nov. 2016. URL: http://www.ros.org/core-components/ (cit. on p. 5).

[@Deu16] Deutsche Gesetzliche Unfallversicherung. *DGUV Information 209-074 Industrieroboter*. Jan. 2016. URL: http://publikationen.dguv.de/dguv/pdf/10002/209-074.pdf (cit. on p. 2).

[@Int16] International Federation of Robotics. *Survey: 1.3 million industrial robots to enter service by 2018*. Feb. 2016. URL: http://www.ifr.org/news/ifr-press-release/survey-13-million-industrial-robots-to-enter-service-by-2018-799/ (cit. on p. 2).

[@Pat16] Patrick Davison at Robotic Industries Association. *How Does the Accident in Germany Affect Industrial Robot Safety?* June 2016. URL: http://www.robotics.org/content-detail.cfm/Industrial-Robotics-News/How-Does-the-Accident-in-Germany-Affect-Industrial-Robot-Safety/content_id/5555 (cit. on p. 2).