The Simox Robotics Grasp Planner Library

LaTeX Pandora@home

Programmers Manual

Kalogiannis Grigorios

Automation and Robotics Laboratory - Pandora@home
$\boxed{\mathcal{PL}}$

Contents

Chapter 1

Introduction

1.1 Robotic grasping

Over the past decades, research in robotic grasping has flourished. Several algorithms have been developed for synthesizing robotic grasps in order to achieve stability,force-closure,task compatibility and other properties. Different approaches have been developed to meet these goals, and substantial improvements have been claimed. Thus,the availability of a large number of algorithms for our purpose has made it difficult to choose, since their approaches and assumptions are different.

1.2 Background and terminology

The basic function of a gripper is to grasp objects and possibly manipulate them by means of its fingers. One of the essential properties looked for in the grasp configuration selection is the immobilization of the grasped object(its equilibrium) against the possible external disturbance.The set of fingers grasping the object by the fingertips can also be seen,from a mechanical point of view, as distributed impedances on the object surface.

Given an object, grasp synthesis refers to the problem of finding a grasp configuration that satisfies a set of criteria relevant for the grasping task. Finding a suitable grasp among the infinite set of candidates is a challenging problem and has been addressed frequently in the robotics community, resulting in an abundance of approaches.

Consider an object grasped at N contact points.At each contact location, the object is subject to normal/tangential forces and torsional moment about the normal. Let us denote theses wrenches by w_n^i, w_t^i, w_θ^i respectively, and their corresponding magnitude by c_n^i, c_t^i, c_θ^i. Each contact may be either frictionless, frictional or soft. In the case of a frictional contact, there are only normal and tangential wrenches. For a friction less contact only the normal wrench is considered. The wrench matrix $\mathbf{W}$ is composed of the mentioned vector wrenches

arranged in columns.Let us donate $\mathbf{C}$ the corresponding wrench magnitude vector.

A grasped object is defined to be inequilibrium if the sum of all forces and the sum of all moments acting on it are equal to zero.An equilibrium grasp may be stable or unstable.The stability was is detailed as follow: A grasped object at equilibrium, in which all forces and moments can be derived from a potential function $V(q)$ is stable if $\forall \triangle q \neq 0, \Delta V > 0$

A grasp verify the force closure property if and only if, for any external wrench $\widehat{\mathbf{w}}$, there exists a magnitude vector λ satisfying the constraint equalities, therefore $\mathbf{W}\ \lambda = \widehat{\mathbf{w}}$

1.3 Analytic Approaches

Analytic approaches refers to methods that construct force-closure grasps with a multi-fingered robotic hand that are dexterous, in equilibrium, stable and exhibit a certain dynamic behavior. Grasp synthesis is then usually formulated as a constrained optimization problem over criteria that measure one or several of these four properties. In this case, a grasp is typically defined by the grasp map that transforms the forces exerted at a set of contact points to object wrenches.The criteria are based on geometric, kinematic or dynamic formulations.

Analytic approaches usually based on assumptions such as simplified contact models, Coulomb friction and rigid body modeling.Although these assumptions render grasp analysis practical, inconsistencies and ambiguities especially regarding the analysis of grasp dynamics are usually attributed to their approximate nature.

1.4 Empirical - Data Driven Approaches

Empirical or data-driven approaches rely on sampling grasp candidates for an object and ranking them according to specific metric. This process is usually based on some existing grasp experience that can be a heuristic or is generated in simulation or on a real robot. In this case grasp is commonly parameterized by

- the grasping point on the object with which the tool center point(TCP) should be aligned,
- the approach vector which describes the 3D angle that the robot hand approaches the grasping point with,
- the wrist orientation of the robotic hand and
- an initial finger configuration

Data-driven approaches differ in how the set of grasp candidates is sampled, how the grasp quality is estimated and how good grasps are represented for future use. Some methods measure grasp quality based on analytic formulations, but more commonly they encode e.g. human demonstrations, perceptual information or semantics.

Chapter 2

Installation

2.1 Get Simox from a PPA

You have the choice of either installing the simox package from our PPA or by compiling the sources (see below). The PPA/package currently is available only on Ubuntu distributions starting from Precise (12.04) up to and including Trusty (14.04).

First, add the PPA to your list of repositories and then install the package via apt-get:

```
sudo add-apt-repository ppa:simox-dev/ppa
sudo apt-get update
sudo apt-get install simox
```

Keep in mind that this will only install simox without any special options enabled (e.g. bullet) and with Coin visualization support.

Get Simox from gitlab repository

Retrieve the sources with:

```
git clone https://gitlab.com/Simox/simox.git
```

Or use a GUI-based GIT client to grab the sources from: 'https://gitlab.com/Simox/simox.git'

2.2 Install dependencies

You will need the following libraries installed on your system:

- CMake >=2.8.3: A cross-platform, open-source build system.http://www.cmake.org/
- Boost >=1.42: Provides free peer-reviewed portable C++ source libraries.http://www.boost.org
- Eigen >=3.0: A header-only C++ template library for linear algebra: matrices, vectors, and numerical solvers.http://eigen.tuxfamily.org

2.2.1 Visualization / 3D Model support

1 Coin3D/Qt (recommended) For 3D visualization support Simox offers a plugin-like mechanism that can be extended in order to make your favorite 3D library usable. In the current version Coin3D/Qt support is already implemented allowing to handle Open Inventor file formats (.iv, .vrml) in combination with Qt-based GUIs. If you want to use this visualization supported you will additionally need the following libraries:

- Qt4 (>=4.6) or Qt5: Cross-platform application and UI framework.
  http://qt-project.org/downloads
- Coin3D (>=3.1.3): A high-level, retained-mode toolkit for effective 3D graphics development.
  https://bitbucket.org/Coin3D/coin
- SoQt (>=1.5): A set of Qt GUI bindings for Coin3D library.
  https://bitbucket.org/Coin3D/soqt

2 OpenSceneGraph With OpenSceneGraph (OSG) a LGPL library can be used with Simox (instaed of the GPL-library Coin3D). OSG supports several file formats for 3D-model loading, e.g. 3dc, 3ds, flt, geo, iv, ive, lwo, md2, obj, osg, whereas some of them depend on external libraries which must be present on your system. Have a look at OSG's installation and plugin help pages.

- Qt4 (>=4.6): Cross-platform application and UI framework. QT versions¿=5 are not yet supported.
  http://qt-project.org/downloads
- OpenSceneGraph (>=3.0): A high performance 3D graphics toolkit.
  http://www.openscenegraph.org

3 No visualization support With this setup, visualization is disabled and hence, no collision detection can be performed. Nevertheless all operations which do not rely on 3d models, e.g. coordinate transformations, are still available.

2.2.2 Linux

Simox is tested on several Ubuntu platforms and there shouldn't be any problems with other distributions

- In order to setup the dependencies, you could either
  - have a look at the package manager of your distribution (maybe the libs you will need are already installed)
  - or you install them by hand, i.e. compile and install the sources.
  - On an Ubuntu 14.04 system you can install the libraries (with coin support) the following way (on other systems the names of the libraries may differ slightly):

- create a build directory, e.g

```
<path_to_your_simox_checkout>/build
```

- Setup your environment variables (not needed if the dependencies were installed to the standard linux system paths, e.g. when the package management system was used)
  - `Eigen3_DIR->Path to Eigen3`
  - `BOOST_ROOT -> The Boost directory`
  - `Coin3D_DIR -> The Coin3D library`
  - `SoQt_DIR -> the SoQt lib (Has to be compiled with the used Coin3D and Qt version)`
  - `QT_QMAKE_EXECUTABLE -> The complete path to the qmake binary file. Usually located at`

    ```
    <QtDir>/bin/qmake
    ```

  - `Simox_DIR -> The path to your simox checkout`
  - `An exemplary setup could look like this (). You can create a simox_setpath.sh file and by calling source simox_setpath.sh all exports are set in your shell environment.`
  - Create make files:
    - go to your build directory: cd a
    - call either ccmake .. or use the cmake-gui .. (see below for further information about cmake-gui)
    - configure ('c' in ccmake) and generate make files ('g' in ccmake)
  - Compile Simox
    - Now Simox with all examples can be compiled by typing 'make'. Optionally you can specify how many processes should be used for compiling (e.g. 'make -j4')
  - IDE
    - You can use the make files within your favorite IDE, like Eclipse http://www.eclipse.org or QtCreator http://qt-project.org/doc/qt-5/topics-app-development.html (which can be used also for non-Qt projects)

```
sudo apt-get install build-essential libboost-all-dev
   libeigen3-dev libsoqt4-dev libcoin80-dev cmake
   cmake-gui libqt4-dev subversion
```

2.3 Grasp planning Tools - SOA

Need of grasping planning tools

Automatic grasp planning is a difficult problem because of the huge number of possible hand configurations. Humans simplify the problem by choosing an appropriate prehensile posture appropriate for the object and task to be performed. By modeling an object as a set of shape primitives (spheres, cylinders, cones and boxes) grasping tools can use a set of rules to generate a set of grasp starting positions and pregrasp shapes. These generated data can then be tested on the object model. Several grasp planning tools offer also the functionality to evaluate and test the resulted grasp in order to present the best grasp to the user.

Graspit! Toolbox

GraspIt! was created to serve as a tool for grasping research. It is a simulator that can accommodate arbitrary hand and robot designs. It can also load objects and obstacles of arbitrary geometry to populate a complete simulation world. The GraspIt! engine includes a rapid collision detection and contact determination system that allows a user to interactively manipulate a robot or an object and create contacts between them. Once a grasp is created, one of the key features of the simulator is the set of grasp quality metrics. Each grasp is evaluated with numeric quality measures, and visualization methods allow the user to see the weak point of the grasp and create arbitrary 3D projections of the 6D grasp wrench space.

GraspIt! features include:

3D user interface allowing the user to see and interact with a virtual world containing robots, objects and obstacles; a library of robotic hand models; computation of numerical grasp quality metrics and visualization methods for the Grasp Wrench Space; grasp planning; dynamics engine; support for Soft Finger Contacts; support for low-dimensional hand posture subspaces (also known as eigengrasps or hand synergies); interaction with hardware and sensors, such as a Barrett hand, Flock of Birds tracker and Cyberglove (Windows only);

OpenRave Toolbox

OpenRAVE provides an environment for testing, developing, and deploying motion planning algorithms in real-world robotics applications. The main focus is on simulation and analysis of kinematic and geometric information related to motion planning. OpenRAVE's stand-alone nature allows is to be easily integrated into existing robotics systems. It provides many command line tools to work with robots and planners, and the run-time core is small enough to be used inside controllers and bigger frameworks. An important target application is industrial robotics automation.

2.4 The Simox Grasp Planner Case

2.4.1 Indroduction - What is Simox

Simox is a lightweight platform independent C++ toolbox containing three libraries for 3D simulation of robot systems, sampling based motion planning and grasp planning.

The Virtual Robot library is used to define complex robot systems which may cover multiple robots with many degrees of freedom. The robot structure and its visualization can be easily defined via XML files and environments with obstacles and objects to manipulate are supported. The libraries Grasp Studio and Saba use these definitions for planning grasps or collision-free motions.

State-of-the-art implementations of sampling-based motion planning algorithms (e.g. Rapidly-exploring Random Trees) are served by the Saba library which was designed for efficient planning in high-dimensional configuration spaces.

The library Grasp Studio offers possibilities to compute grasp quality scores for generic end-effector definitions (e.g. a humanoid hand). The implemented 6D wrench-space computations, offer the possibility to easily (and quickly) measure the quality of an applied grasp to an object. Furthermore, the implemented planners are able to generate grasp maps for given objects automatically.

2.4.2 Components of Simox

VirtualRobot: The robot library

The library VirtualRobot can be used to define complex robot systems, which may cover multiple robots with many degrees of freedom. The robot structure and its visualization can be easily defined via XML files and environments with obstacles and objects to manipulate are supported. Further, basic robot simulation components, as Jacobian computations and generic Inverse Kinematics (IK) solvers, are offered by the library. Beyond that, extended features like tools for analyzing the reachable workspace for robotic manipulators or contact determination for grasping are included.

Saba: The motion planning library

With Saba, a library for planning collision-free motions is offered, which directly incorporates with the data provided by VirtualRobot. The algorithms cover state-of-the-art implementations of sampling-based motion planning approaches (e.g. Rapidly-exploring Random Trees) and interfaces that allow to conveniently implement own planners. Since Saba was designed for planning in high-dimensional configuration spaces, complex planning problems for robots with a high number of degrees of freedom (DoF) can be solved efficiently.

Grasp Studio: The grasp planning library

The Grasp Studio library contains methods and tools used for measuring grasp qualities. Therefore an interface to the qhull library is provided to build convex hulls in 3D or 6D. The methods for measuring grasp qualities implement methods based on force-space related algorithms in 3D and a full implementation of 6D grasp wrench spaces is included. Grasping setups of simple end-effectors, multi-finger hands, multi-hand and multi-robot grasps can be evaluated by the Grasp Studio library. Grasp planners are implemented for building object specific grasp maps for given end-effectors.

2.4.3 Inside the VirtualRobot library

With the Virtual Robot library robots and environments can be specified and accessed via XML definitions. The structure of a robot is given by its joints (called RobotNodes) which are defined via Denavit-Hartenberg conventions or an easy-to-use "Translation+Rotation" scheme. The robot specification holds further information as name, joint limits, sensors, visualization and collision models. For convenient access kinematic chains and collections of collision models can be additionally defined. An instance of such a robot definition can be visualized by a viewer to show the current state of a robot. Furthermore, environments, obstacles and objects to manipulate can be defined and visualized. Thus it is possible to construct complex scenes used for simulation or planning and to store and load such scene definitions to/from XML files. The main features of the library are:

- The library defines an interface for collision checking, allowing to exchange the collision checker used for determining distances or collisions. Currently the PQP collision checker is used due to its fast and robust implementation.
- A generic computation of Jacobains and its Pseudoinverse are offered for arbitrary kinematic chains.
- A generic IK solver, utilizing randomized approaches, is implemented for solving the IK for redundant kinematic chains.
- Reachability distributions are supported, in order to precompute the 6D reachability of a kinematic structure (e.g. an arm) and to quickly decide whether a pose in workspace is reachable or not.
- A generic definition of end effectors allows to realize complex hands (e.g. humanoid hands can be defined). End effectors can be opened/closed and contact information can be retrieved e.g. for grasp scoring.
- It is possible to store and load object-related grasping information for a given end-effector, which directly can be used within the IK solvers.
- Multithreading is supported, e.g. multiple instances of the collsiion checker can be used to parallelize collision queries.

Build a robot/environment model

Robot models are defined via XML files, where the kinematic structure as well as additional information (e.g. visualizations, joint definitions or physics information) is stored. A robot consists of so-called Robot Nodes, which hold joint definitions and visualization data. After loading, these data structures can be accessed through the VirtualRobot library.

In the following examples we start with a simple robot system in order to show how the XML definitions are structured. Later more complex examples are given, where you can see how advanced robot definitions can be created.

Example

This minimal example shows how to define a robot. The file can be found in the doc/tutorial folder of the simox source tree. It can be loaded with VirtualRobot's RobotViewer example (located at VirtualRobot/examples/RobotViewer).

```
<Robot Type="DemoRobot" RootNode="root">

   <RobotNode name="root">
   </RobotNode>

</Robot>
```

Visualizations

Now we add a visualization. In the following example, the Coin3D/Qt visualization support is used, indicated by the type="Inventor" attribute of the visualization's File tag (in case OpenSceneGraph is used for 3D-model importing and visualization, the attribute would be type="osg"). The location of the joint.iv file is given relatively to the location of the XML file, so that the directory iv is searched for the file joint.iv.

```
<Robot Type="DemoRobot" RootNode="root">

   <RobotNode name="root">
       <Visualization>
           <File type="Inventor">iv/joint.iv</File>
    </Visualization>
   </RobotNode>

</Robot>
```

The joint.iv file defines a cylinder in the OpenInventor format. As specified in OpenInventor, the cylinder is aligned with the y axis and it is 500 mm long. In order to set one side of the cylinder to the (local) origin, the object is moved by 250 mm.

```
#Inventor V2.0 ascii
Separator {
   Translation {
       translation 0 250 0
   }
   Cylinder {
       radius 40
       height 500
       parts (SIDES | TOP | BOTTOM)
   }
}
```

Joints

In this example the robot gets a movable joint. The Joint type is specified with the revolute attribute and the rotation axis. The Transform tag specifies a fixed transformation that is applied before the joint. Hence, a RobotNode may imply two transformations:

- Transform This fixed transformation is applied to the parent's global pose.
- Joint: This transformation depends on the joint value and the resulting pose specifies the location where the visualization is linked to. This final pose of the RobotNode is noted as GlobalPose in Simox. This is the starting point for computing the coordinate systems of all children.

```
<Robot Type="DemoRobot" RootNode="root">

  <RobotNode name="root">
    <Child name="joint␣1"/>
  </RobotNode>

  <RobotNode name="joint␣1">
    <Transform>
      <Translation x="0" y="500" z="0"/>
    </Transform>
    <Joint type="revolute">
      <Axis x="0" y="0" z="1"/>
    </Joint>
    <Visualization>
      <File type="Inventor">iv/joint.iv</File>
    </Visualization>
  </RobotNode>

</Robot>
```

Kinematic Chains

Now we build a longer kinematic chain by connecting multiple RobotNodes. Some of the nodes are fixed (no Joint tag), while others can be moved indicated by a ¡Joint type=”revolute’¿ definition.

```
<Robot Type="DemoRobot" RootNode="root">

   <RobotNode name="root">
     <Child name="joint␣1"/>
   </RobotNode>

   <RobotNode name="joint␣1">
     <Joint type="revolute">
       <Axis x="0" y="0" z="1"/>
     </Joint>
     <Visualization>
       <File type="Inventor">iv/joint.iv</File>
     </Visualization>
     <Child name="node␣1"/>
    </RobotNode>

    <RobotNode name="node␣1">
      <Transform>
        <Translation x="0" y="500" z="0"/>
      </Transform>
      <Visualization>
        <File type="Inventor">iv/ball.iv</File>
      </Visualization>
      <Child name="joint␣2"/>
    </RobotNode>

    <RobotNode name="joint␣2">
      <Joint type="revolute">
        <Axis x="0" y="1" z="0"/>
      </Joint>
      <Visualization>
        <File type="Inventor">iv/joint2.iv</File>
      </Visualization>
      <Child name="node␣2"/>
    </RobotNode>

    <RobotNode name="node␣2">
      <Transform>
        <Translation x="0" y="0" z="500"/>
      </Transform>
      <Visualization>
```

```
        <File type="Inventor">iv/ball.iv</File>
      </Visualization>
      <Child name="joint 3"/>
    </RobotNode>

    <RobotNode name="joint 3">
      <Joint type="revolute">
        <Axis x="1" y="0" z="0"/>
      </Joint>
      <Visualization>
        <File type="Inventor">iv/joint.iv</File>
      </Visualization>
    </RobotNode>
</Robot>
```

Denavit-Hartenberg Parameters

A joint can also be definied with Denavit Hartenberg paramters. Therefore the DH tag with the two translational attributes a, d and with both rotational attributes alpha and theta can be used. Please note that you have to specify the units in which the rotation is given (degree or radian). Further, the Limits tag is used to specify the lower and upper joint limits.

```
<Robot Type="DemoRobot" RootNode="root">

    <RobotNode name="root">
        <Child name="joint 1"/>
    </RobotNode>

    <RobotNode name="joint 1">
        <Joint type="revolute">
        <Limits lo="0" hi="120" units="degree"/>
        </Joint>
        <Visualization>
            <File type="Inventor">iv/joint2.iv</File>
        </Visualization>
        <Child name="node 1"/>
    </RobotNode>

    <RobotNode name="node 1">
        <Transform>
            <DH a="0" d="500" alpha="-90" theta="0"
               units="degree"/>
        </Transform>
        <Visualization>
            <File type="Inventor">iv/ball.iv</File>
```

```
        </Visualization>
        <Child name="joint 2"/>
    </RobotNode>

    <RobotNode name="joint 2">
        <Joint type="revolute">
            <Limits lo="-45" hi="45" units="degree"/>
        </Joint>
        <Visualization>
            <File type="Inventor">iv/joint2.iv</File>
        </Visualization>
        <Child name="node 2"/>
    </RobotNode>

    <RobotNode name="node 2">
        <Transform>
            <DH a="0" d="500" alpha="90" theta="0"
               units="degree"/>
        </Transform>
    </RobotNode>
</Robot>
```

Coordinate Systems

A RobotNode may cover several transformations.

- Transform: This is a fixed transformation which is applied to the initial pose that is passed by the parent of this node.
- Joint: This transformation depends on the joint type and the current joint value. In classical robot systems this transformation applies a rotation....

The GlobalPose of a RobotNode is always the pose that is retrieved after applying all transformations.

Collision Models

In the following example a finger is modeled with DH parameters. Additionally to the 3D models, the CollisionModel tag is used to specify visualization files that should be used for collision detection. The CollisionModels will automatically be used whenever collision detection is performed, hence usually reduced models are used in order to speed up the collision detection. If the visualization models should be used for collision detection as well you can use the ¡UseAsCollisionModel/¿ tag within a ¡VisualizationModel¿ definition, in order to avoid loading the 3D model twice. Further, bounding boxes can be automatically created to be used as CollisionModels. Therefore you have to use the attribute boundingbox=”true” within the CollisionModel's File tag.

```
<Robot Type="DemoRobot" RootNode="root">

  <RobotNode name="root">
    <Child name="Finger␣Joint1"/>
  </RobotNode>

  <RobotNode name="Finger␣Joint1">
    <Joint type="revolute">
      <Limits unit="degree" lo="-25" hi="25"/>
    </Joint>
    <Visualization enable="true">
      <File type="Inventor">iv/LeftIndexFinger/
          leftIndexFinger0.iv</File>
    </Visualization>
    <CollisionModel>
      <File type="Inventor">iv/LeftIndexFinger/
          col_leftIndexFinger0.iv</File>
    </CollisionModel>
    <Child name="BaseToFinger"/>
  </RobotNode>

  <RobotNode name="BaseToFinger">
    <Transform>
      <DH a="14.8" d="0" theta="0" alpha="90" units="
          degree"/>
    </Transform>
    <Child name="Finger␣Joint2"/>
  </RobotNode>

  <RobotNode name="Finger␣Joint2">
    <Transform>
      <DH a="0" d="1.75" theta="0" alpha="0" units="
          degree"/>
    </Transform>
    <Joint type="revolute">
      <Limits unit="degree" lo="0" hi="90"/>
    </Joint>
    <Visualization enable="true">
      <File type="Inventor">iv/LeftIndexFinger/
          leftIndexFinger1.iv</File>
    </Visualization>
    <CollisionModel>
      <File type="Inventor">iv/LeftIndexFinger/
          col_leftIndexFinger1.iv</File>
    </CollisionModel>
```

```
    <Child name="Finger Joint3"/>
  </RobotNode>

  <RobotNode name="Finger Joint3">
    <Transform>
      <DH a="25.9" d="0" theta="0" alpha="0" units="
          degree"/>
    </Transform>
    <Joint type="revolute">
      <Limits unit="degree" lo="0" hi="90"/>
    </Joint>
    <Visualization enable="true">
      <File type="Inventor">iv/LeftIndexFinger/
          leftIndexFinger2.iv</File>
    </Visualization>
    <CollisionModel>
      <File type="Inventor">iv/LeftIndexFinger/
          col_leftIndexFinger2.iv</File>
    </CollisionModel>
    <Child name="Finger Joint4"/>
  </RobotNode>

  <RobotNode name="Finger Joint4">
    <Transform>
      <DH a="22" d="0" theta="0" alpha="0" units="
          degree"/>
    </Transform>
    <Joint type="revolute">
      <Limits unit="degree" lo="0" hi="90"/>
    </Joint>
    <Visualization enable="true">
      <File type="Inventor">iv/LeftIndexFinger/
          leftIndexFingerTip.iv</File>
    </Visualization>
    <CollisionModel>
      <File type="Inventor">iv/LeftIndexFinger/
          col_leftIndexFingerTip.iv</File>
    </CollisionModel>
  </RobotNode>

</Robot>
```

Multiple Files

When building complex robot models, usually several parts of the robot can be separated. In order to reflect this fact, robots can be assembled from multiple files. In the following example a robot imports another robot and connects the root node to Joint1. Further, you can see how coordinate axis visualizations are enabled and that a RobotNode can have multiple children, allowing to build tree-like kinematic structures.

```
<Robot Type="robot1" RootNode="Joint1">

    <RobotNode name="Joint1">
        <Joint type="revolute">
            <Limits unit="degree" lo="-90" hi="45"/>
        </Joint>
        <!-- Imports the kinematic structure form
           otherRobot.xml and adds otherRobot's␣
           rootnode␣as␣a␣child␣to␣this␣node␣-->
␣␣␣␣␣␣␣␣<ChildFromRobot>
␣␣␣␣␣␣␣␣␣␣␣␣<File␣importEEF="true">robot_howto7b.xml</
    File>
␣␣␣␣␣␣␣␣</ChildFromRobot>
␣␣␣␣␣␣␣␣<Child␣name="node2.1"/>
␣␣␣␣</RobotNode>

␣␣␣␣<RobotNode␣name="node2.1">
␣␣␣␣␣␣␣␣<Transform>
␣␣␣␣␣␣␣␣␣␣␣␣<DH␣a="0"␣d="500"␣theta="-30"␣alpha="0"␣
    units="degree"/>
␣␣␣␣␣␣␣␣</Transform>
␣␣␣␣</RobotNode>

</Robot>
```

The second file (robot howto7b.xml) looks like this. The file can also be used as a independent robot model.

```
<Robot Type="robot2" RootNode="node1">

  <RobotNode name="node1">
    <Child name="node2"/>
    <Child name="node3"/>
  </RobotNode>

  <RobotNode name="node2">
    <Transform>
        <Translation x="0" y="500" z="0"/>
```

```
    </Transform>
    <Visualization enable="true">
      <CoordinateAxis type="Inventor" enable="true"
         scaling="1" text="TCP1"/>
    </Visualization>
  </RobotNode>

  <RobotNode name="node3">
    <Transform>
        <Translation x="0" y="-500" z="0"/>
    </Transform>
    <Visualization enable="true">
      <CoordinateAxis type="Inventor" enable="true"
         scaling="1" text="TCP2"/>
    </Visualization>
  </RobotNode>

 </Robot>
```

Robot Node Sets

In VirtualRobot so-called RobotNodeSets are used to group joints or to define kinematic chains. A robot node set consists of a set of robot nodes and optionally

- A root node, defining the topmost node which has to be updated in order to internally re-compute the kinematic structure, e.g. when changing joint values. If not given, the root of the robot is used.
- A RobotNode defining the Tool Center Point (TCP) coordinate system. This node specifies a TCP for a kinematic chain. The following example shows how a RobotNodeSet is defined. Please note that the order of the joint definitions is preserved for later use.

```
<Robot Type="DemoRobot" RootNode="root">

   <RobotNode name="root">
     <Child name="joint␣1"/>
   </RobotNode>

   <RobotNode name="joint␣1">
     <Joint type="revolute">
        <Axis x="0" y="0" z="1"/>
     </Joint>
     <Visualization>
       <File type="Inventor">iv/joint.iv</File>
     </Visualization>
```

```
    <Child name="node 1"/>
  </RobotNode>

 <RobotNode name="node 1">
    <Transform>
        <Translation x="0" y="500" z="0"/>
    </Transform>
    <Visualization>
      <File type="Inventor">iv/ball.iv</File>
    </Visualization>
    <Child name="joint 2"/>
  </RobotNode>

  <RobotNode name="joint 2">
    <Joint type="revolute">
      <Axis x="0" y="1" z="0"/>
    </Joint>
    <Visualization>
      <File type="Inventor">iv/joint2.iv</File>
    </Visualization>
    <Child name="node 2"/>
  </RobotNode>

  <RobotNode name="node 2">
      <Transform>
          <Translation x="0" y="0" z="500"/>
      </Transform>
      <Visualization>
          <File type="Inventor">iv/ball.iv</File>
      </Visualization>
      <Child name="joint 3"/>
  </RobotNode>

  <RobotNode name="joint 3">
      <Joint type="revolute">
        <Axis x="1" y="0" z="0"/>
      </Joint>
      <Visualization>
        <File type="Inventor">iv/joint.iv</File>
      </Visualization>
      <Child name="tcp"/>
  </RobotNode>

  <RobotNode name="tcp">
      <Transform>
        <Translation x="0" y="500" z="0"/>
```

```
        </Transform>
        <Visualization>
          <File type="Inventor">iv/ball.iv</File>
        </Visualization>
    </RobotNode>

    <RobotNodeSet name="my kinematic chain"
       kinematicRoot="joint1" tcp="tcp">
      <Node name="joint 1"/>
      <Node name="joint 2"/>
      <Node name="joint 3"/>
    </RobotNodeSet>

</Robot>
```

End Effector Definitions

When complex robot systems, such as humanoid robots, should be modeled a logical definition of their end-effectors is helpful especially in the context of grasp and manipulation planning. In order to access the robot's end-effector conveniently it can be defined in the robot's XML definition. An end-effector is a logical collection of static RobtoNodes and so-called actors which are lists of RobotNodes defining a finger. Please note that the RobotNodes have to be defined within the robot's XML definition. The static parts are not moved when the end-effector is accessed (e.g. closed or opened), whereas the actors are moved. As shown in the example below, actors consist of a set of RobotNodes and for each node several options can be specified:

- name: The name string of the RobotNode. This node must be present in the robot's definition.
- considerCollisions: Here the behavior for collision detection can be adjusted. Possible options are
  - all (standard): Collisions between the static part and the RobotNode and between other actors and the RobotNode are considered.
  - none The node is not considered for collision detection. E.g. when closing the hand an actor will not stop if this node is in collision.
  - actors: Collisions between the static part of teh hand and the node are not considered, but collisions between other actors and the RobotNode are.
  - static: Only collisions between this node and the static part of the end-effector are considered, while collision between this node and other actors are not considered.

- direction: Here the moving direction can be inverted by setting this value to -1. Further, the relative speed of moving can be adjusted (the standard value is 1).

```
<Robot Type="iCub Left Hand" RootNode="root">
    ...
    <Endeffector name="Left Hand" base="Left Hand Init
       " tcp="Left Arm TCP" gcp="Left Arm GCP">
        <Preshape name="Grasp Preshape">
            <Node name="Left Hand Thumb Joint1" unit="
               radian" value="1.3"/>
        </Preshape>

        <Static>
            <Node name="Left Hand Palm"/>
            <Node name="Left Hand Thumb Joint1"/>
        </Static>

        <Actor name="Left Hand Thumb">
            <Node name="Left Hand Thumb Joint2"
               considerCollisions="Actors"/>
            <Node name="Left Hand Thumb Joint3"
               considerCollisions="All"/>
            <Node name="Left Hand Thumb Joint4"
               considerCollisions="All"/>
        </Actor>
        <Actor name="Left Hand Index">
            <Node name="Left Hand Index Joint2"
               considerCollisions="Actors"/>
            <Node name="Left Hand Index Joint3"
               considerCollisions="All"/>
            <Node name="Left Hand Index Joint4"
               considerCollisions="All"/>
        </Actor>
        <Actor name="Left Hand Middle">
            <Node name="Left Hand Middle Joint2"
               considerCollisions="Actors"/>
            <Node name="Left Hand Middle Joint3"
               considerCollisions="All"/>
            <Node name="Left Hand Middle Joint4"
               considerCollisions="All"/>
        </Actor>
        <Actor name="Left Hand Ring">
            <Node name="Left Hand Ring Joint2"
               considerCollisions="Actors"/>
```

```
            <Node name="Left Hand Ring Joint3"
                considerCollisions="All"/>
            <Node name="Left Hand Ring Joint4"
                considerCollisions="All"/>
        </Actor>
        <Actor name="Left Hand Pinky">
            <Node name="Left Hand Pinky Joint2"
                considerCollisions="Actors"/>
            <Node name="Left Hand Pinky Joint3"
                considerCollisions="All"/>
            <Node name="Left Hand Pinky Joint4"
                considerCollisions="All"/>
        </Actor>
    </Endeffector>
</Robot>
```

Objects and Scenes

Scenes can be defined via XML files, allowing to specify a setup for robots and obstacles. Further, configurations can be defined allowing to easily access them later on. The following tags can be used:

- Robot: Define a robot with a name and an optional initial configuration initConfig.
  - The File tag specifies the robot's XML file from which the model should be loaded.
    - Files are searched in all dataPaths which are specified in VirtualRobot::RuntimeEnvironment (initially some simox-related standard data paths are set).
    - Optional the attribute path can set to absolute or relative, in order to force absolute or relative (w.r.t. the scene file) path names.
  - Multiple Configurations can be defined, which can be accessed via their name strings.
    - A Configuration is a list of RobotNodes with corresponding joint values.
  - A GlobalPose tag encapsulates a Transfrom tag which can be used to position the robot in the scene.
- Obstacles can be loaded from files. Here the same syntax is provided as when setting Visualization tags for a RobotNode:
  - The type of 3d model must be specified in the File tag. Again the file names are searched in the paths, defined in VirtualRobot::RuntimeEnvironment.

Optionally the attributes path='absolute' or path='relative' allow to specify absolute or scene-file-relative file names.

  - A GlobalPose can be used to position the object.
  - The ¡UseAsCollisionModel/¿ tag ensures that the visualization model is used as a collision model.

- A ManipulationObject is basically a 3d object with predefined grasping information. These objects can also be positioned via the GlobalPose tag.

```
<Scene name="Armar3Scene">

    <Robot name="Armar3" initConfig="start">
        <File>robots/ArmarIII/ArmarIII.xml</File>
        <Configuration name="start">
            <Node name="Shoulder 1 L" unit="radian"
               value="-0.85"/>
            <Node name="Shoulder 2 L" unit="radian"
               value="-0.8"/>
            <Node name="Upperarm L" unit="radian"
               value="-0.85"/>
            <Node name="Shoulder 1 R" unit="radian"
               value="-0.85"/>
            <Node name="Shoulder 2 R" unit="radian"
               value="0.8"/>
            <Node name="Upperarm R" unit="radian"
               value="0.85"/>
        </Configuration>
        <GlobalPose>
            <Transform>
                <Translation x="1000" y="0" z="0"/>
            </Transform>
        </GlobalPose>
    </Robot>

    <Robot name="Armar3b">
        <File>robots/ArmarIII/ArmarIII.xml</File>
    </Robot>

    <Obstacle name="Box">
        <Visualization>
            <File type="Inventor">objects/iv/
               box1000x500x300.iv</File>
            <UseAsCollisionModel/>
        </Visualization>
        <GlobalPose>
```

```
            <Transform>
                <Translation x="9000" y="1500" z="150"
                    />
                <rollpitchyaw units="degree" roll="0"
                    pitch="0" yaw="90"/>
            </Transform>
        </GlobalPose>
    </Obstacle>

    <ManipulationObject name="Plate">
        <File>objects/plate.xml</File>
        <GlobalPose>
            <Transform>
                <Translation x="-250" y="0" z="600"/>
                <rollpitchyaw units="degree" roll="90"
                     pitch="0" yaw="90"/>
            </Transform>
        </GlobalPose>
    </ManipulationObject>

</Scene>
```

Virtual Robot: The robot library

Once a robot and/or scene information are created , one can load and access the data via the VirtualRobot library. The C++ API offers convenient access, which is described in the following sections.

Obstacles and Grasps

Simple obstacles can be created as follows:

```
VirtualRobot::ObstaclePtr box = VirtualRobot::Obstacle
    ::createBox(30.0f,30.0f,30.0f); // in mm
```

They can be positioned via homogeneous matrices:

```
box->setGlobalPose(matrix4x4);
```

Obstacles that can be grasped are called *ManipulationObjects* and offer the (optional) possibility to define grasps for a specific robot/end effector.

```
VirtualRobot::ManipulationObjectPtr object =
    VirtualRobot::ObjectIO::loadManipulationObject("
    filename.xml");
```

The corresponding grasps can be retrieved with:

```
VirtualRobot::GraspSetPtr grasps = object->getGraspSet
    ("myRobot", "myEEF");
```

A grasp can be accessed as follows:

```
VirtualRobot::GraspPtr g = grasps->getGrasp(0);
VirtualRobot::GraspPtr g2 = grasps->getGrasp("name␣of␣
    grasp");
```

A grasp basically defines a object related pose of the end effector, whereas the pose of the end effector is given in its TCP coordinate system.

To get the resulting pose for the TCP, when considering the object at pose mObject, you can use the following method:

```
Eigen::Matrix4f mTCP = g->getTcpPoseGlobal(mObject);
```

If you want to compute the pose that an object has to be set to, in order that the grasp g can be applied, you need to pass a robot, which is used to retrieve the current configuration:

```
Eigen::Matrix4f mObject = g->getTargetPoseGlobal(robot
    );;
```

Scenes are a collection of robots, obstacles, and ManipulationObjects. A scene can be loaded and accessed in the following way:

```
VirtualRobot::ScenePtr scene = VirtualRobot::SceneIO:
    loadScene("scenefile.xml");

// access robots
std::vector<VirtualRobot::RobotPtr> robots = scene->
    getRobots();
VirtualRobot::RobotPtr robot = scene->getRobot("name␣
    of␣robot");

// get all configurations for a specific robot
 std::vector<VirtualRobot::RobotConfigPtr > configs =
     scene->getRobotConfigs(robot);

// get all Obstacles/ManipulationObjects
std::vector<VirtualRobot::ManipulationObjectPtr>
    manipObjects = scene->getManipulationObjects();
std::vector<VirtualRobot::ObstaclePtr> obstacles =
    scene->getObstacles();

// get sets of SceneObjects (== obstacles or
    manipulationobjects)
```

```
std::vector<VirtualRobot::SceneObjectSetPtr>
    objectSets = scene->getSceneObjectSets();
```

Collision Detection

Collision Detection is a crucial part of any sampling-based motion planning algorithm. Hence, Simox takes special care of efficiently handling collision and distance queries.

Collision Engine

Simox allows to exchange the collision detection engine. By default a slightly updated version of the efficient and reliable library PQP is used. The sources are shipped with simox and the compilation process is handled by cmake, so there is no need of installing PQP on your own. An advantage of PQP is, that no prerequisites have to be met by the model representation as long as they exist as a set of triangles. PQP handles unordered sets of triangles with high efficiency.

If you intend to use a different collision engine have a look at the sources in VirtualRobot/CollisionDetection.

Collision Checker

In Simox a global collision checker singleton exists that can be accessed as follows:

```
VirtualRobot::CollisionCheckerPtr collisionChecker =
    VirtualRobot::CollisionChecker::
    getGlobalCollisionChecker();
```

Collision Checker

Collision Models All models (Objects, Obstacles, Robots, RobotNodes, etc) in Simox have two 3D representations. One for visualization (high def) and one for collision detection purposes (low def). You can visualize the models as well as the internal triangulated data structure by passing teh corresponding VirtualRobot::SceneObject::VisualizationType flag (Full, Collison, CollisionData) to the visualization routines. The collision models are usually defined in the robot's or object's XML description file. You can access them with:

```
VirtualRobot::CollisionModelPtr colModel = object->
    getCollisionModel();
```

Collision and Distance Queries The collision checker can be asked if two models are in collision or not:

```
bool inCollision = collisionChecker->checkCollision(
    colModel1,colModel2);
```

Additionally, the shortest distance between the two surfaces can be queried. The following code fragment shows how to get the shortest distance and the corresponding points on the object's surface:

```
Eigen::Vector3f surface1,surface2;
float dist = collisionChecker->calculateDistance(
    colModel1,colModel2,surface1,surface2);
```

Instead of using the collision models, SceneObjects (and all derived objects like Obstacles and RobotNodes) can be directly passed to the collision checker:

```
VirtualRobot::ObstaclePtr obstacle = VirtualRobot::
    Obstacle::createBox(100.0f,100.0f,100.0f);
obstacle->setGlobalPose(Eigen::matrix4f::Identity());
VirtualRobot::RobotNodePtr robotNode = robot->
    getRobotNode("Joint1");
bool inCollision = collisionChecker->checkCollision(
    robotNode,obstacle);
```

When operating with sets of objects (like kinematic chains, or a bunch of obstacles) you can use SceneObjectSets and RobotNodeSets as well:

```
VirtualRobot::ObstacleSetPtr obstacles = scene->
    getSceneObjectSet("All␣Obstacles");
VirtualRobot::RobotNodeSetPtr kinChain = robot->
    getRobotNodeSet("Left␣Arm");
float minDist = collisionChecker->calculateDistance(
    kinChain,obstacles);
```

Complex Collision Setups Complex collision queries can be handled with the VirtualRobot::CDManager class. Here multiple pairs of object(s) can be specified which all should be checked for validity. This can be useful when multiple parts of a robot have to be considered, e.g. for motion planning.

In the following example an obstacle and a robot node are selected for collision detection. Additionally any self collisions between the kinematic structures of the arm and the torso and head should be considered.

```
VirtualRobot::CDManagerPtr cdm(new VirtualRobot::
    CDManager());
cdm->addCollisionPair(obstacle,robotNode);
cdm->addCollisionPair(rnsArm,rnsTorso);
cdm->addCollisionPair(rnsArm,rnsHead);
bool inCollision = cdm->isInCollision();
```

Workspace Analysis: Reachability and Manipulability

representation of the spatial reachability of a robot's manipulator can be precomputed in order to efficiently search for reachable grasps/poses during online processing. To this end a 6D voxel grid is filled, either with binary values (reachable/not reachable) or with quality information that indicates how well a pose can be reached. E.g. the manipulability can be used as such a quality measure.

From the base class for performing workspace analysis WorkspaceRepresentation, currently two derivations exist:

- Reachability: This implementation can be used to encode reachable 6D poses with a simple quality measure which is related to the volume in joint space that maps to a voxel.
- Manipulability: This class encapsulates an enhanced workspace representation that uses a basic manipulability measure for determining the quality of robot poses. Custom implementations of quality measures can be used as well.
- Reachability Maps are useful to determine promising robot base positions for grasping. Based on a reachability distribution of the manipulator, the reachability map can becomputed as a discretized 2D distribution serving potential robot positions for reaching a grasping pose.

Jacobians and IK solving

Simox provides a generic IK solver, which is based on the Jacobian's Pseudoinverse approach. The IK solver is instantiated with a kinematic chain and it can be queired for 3D or 6D poses. The GenericIKDemo and JacobiDemo provide exemplary implementations. Here is a short example:

```
RobotNodeSetPtr kc = robot->getRobotNodeSet("Left␣Arm"
    );
// several Jacobi inversions algorithms are available
// here, the SVD damped approach is used
GenericIKSolverPtr ikSolver(new GenericIKSolver(kc,
    JacobiProvider::eSVDDamped));

// solve a query (only position)
Eigen::Matrix4f pose = kc->getTCP()->getGlobalPose();
pose(0,3) += 10.0f; // move 10 mm
bool ok = ikSolver->solve(pose,IKSolver::Position);

// solve a query (position and orientation)
// setup stepsize and max gradient decent loops
// lower stepsize -> better convergence but slower
// max loops: Just needed in special cases. Internally
    the iterative process is
// canceled when no progress has been made in the last
    step
ikSolver->setupJacobian(0.3f, 30);
ok = ikSolver->solve(pose,IKSolver::All);

// Search solution to one of the grasps of the
    ManipulationObject
```

```
// current pose of object is considered
ok = ikSolver->solve(object, IKSolver::All);

// Search solution for one specific grasp applied to
    the object
GraspPtr g = object->getGraspSet("LeftHand")->getGrasp
    ("Grasp0");
ok = ikSolver->solve(object, g, IKSolver::All);

// By default, the current configuration of the robot
    is used as start pose
// If that failes (e.g. for large distances), the
    IKSolver can generate random seed poses
// for the iterative gradient decent approach (50 in
    the following example).
ok = ikSolver->solve(pose, IKSolver::Position, 50);
```

Hierarchical IK solving

Complex robot systems usually have to meet multiple objectives for IK solving. Such objectives could include stability, feet postures, eef positions, etc. Simox provides methods for hierarchical IK solving, where a stack of tasks is processed by solving each task in the Null Space of the preceding tasks. Therefore JoacobiProviders are implemented for several objectives like pose reaching and stability. The framework is extendable and allows to implement custom JacobiProvides in order to combine them for hierarchical IK solving.

```
HierarchicalIKPtr hik(new HierarchicalIK(rnsJointsAll)
    );
std::vector<HierarchicalIK::JacobiDefinition> jacobies
    ;

// first Right feet pose
DifferentialIKPtr ikLeft2Right(new DifferentialIK(
    rnsJointsAll));
Eigen::Matrix4f trafoLeft2Right = feetPosture->
    getTransformationLeftToRightFoot();
Eigen::Matrix4f goalRight = feetPosture->getLeftTCP()
    ->getGlobalPose() * trafoLeft2Right;
ikLeft2Right->setGoal(goalRight,feetPosture->
    getRightTCP());
HierarchicalIK::JacobiDefinition jd;
jd.jacProvider = ikLeft2Right;
jacobies.push_back(jd);

// second: COM
```

```
CoMIKPtr comIK(new CoMIK(rnsJointsAll,rnsModelsAll));
supportPolygon->updateSupportPolygon(10.0f);
Eigen::Vector2f c = MathTools::getConvexHullCenter(
   supportPolygon->getSupportPolygon2D());
comIK->setGoal(c);
HierarchicalIK::JacobiDefinition jd2;
jd2.jacProvider = comIK;
jacobies.push_back(jd2);

// third: EEF pose
Eigen::Matrix4f goal = tcpGoal;
DifferentialIKPtr ikTCP(new DifferentialIK(
   rnsJointsAll));
ikTCP->setGoal(goal,tcp);
HierarchicalIK::JacobiDefinition jd3;
jd3.jacProvider = ikTCP;
jacobies.push_back(jd3);

//compute step
Eigen::VectorXf delta = hik->computeStep(jacobies,
   stepsize);
Eigen::VectorXf jv(delta.rows());
rnsJointsAll->getJointValues(jv);
jv += delta;
rnsJointsAll->setJointValues(jv);
```

Summarizing Virtual Robot Library

2.4.4 Inside the Grasp Studio

GraspStudio offers possibilities to compute the grasp quality for generic end-effector definitions, e.g. a humanoid hand. The implemented 6D wrench-space computations can be used to easily (and quickly) determine the quality of an applied grasp to an object. Furthermore, the implemented planners are able to generate grasp maps for given objects automatically.

GraspStudio is a library that closely incorporates with VirtualRobot and which can be used for efficient grasp planning. Based on a robot defined in VirtualRobot, any end effector can be decoupled from the model and considered for grasp planning.

The Grasp Center Point (GCP) of an end effector defines the favorite grasping position and an approach direction. A grasp planner interface is provided, which is used for the implementation of a generic grasp planner. This generic grasp planner mainly relies on two exchangeable functionalities: A generator for building grasping hypothesis and a grasp evaluation component.

The Generic Grasp Planner

In the following setup, it is showed how the generic grasp planner can be setup and used for planning feasible grasps.

```
// load robot
VirtualRobot::RobotPtr robot = VirtualRobot::RobotIO::
    loadRobot(filename);

// get end effector
VirtualRobot::EndEffectorPtr eef = robot->
    getEndEffector("Left Hand");

//  set eef to a preshape configuration which is
    specified in the eef's XML defintion
eef->setPreshape("Grasp Preshape");

// load object
VirtualRobot::ManipulationObjectPtr object =ObjectIO::
    loadManipulationObject("MashedPotatoes.xml");
```

Setup the Grasp Planner

To setup the grasp planner, firstly a quality measure module (GraspQualityMeasureWrenchSpace) and a grasp hypotheses generator (ApproachMovementSurfaceNormal) are built. These objects are passed to the generic grasp planner together with some parameters.

```
GraspStudio::GraspQualityMeasureWrenchSpacePtr
    qualityMeasure(new GraspStudio::
    GraspQualityMeasureWrenchSpace(object));
qualityMeasure->calculateObjectProperties();
GraspStudio::ApproachMovementSurfaceNormalPtr approach
    (new GraspStudio::ApproachMovementSurfaceNormal(
    object,eef));

// get the decoupled eef, which is an independent
    VirtualRobot::RobotPtr
VirtualRobot::RobotPtr eefCloned = approach->
    getEEFRobotClone();

// newly created grasps will be stored here
VirtualRobot::GraspSetPtr grasps(new VirtualRobot::
    GraspSet("my new grasp set",robot->getType(),eef->
    getName()));
```

```
// setup the planner: The minimum quality that must be
    reached by a planned grasp is set to 0.2 and the
    grasps have to be force-closure
GraspStudio::GenericGraspPlannerPtr planner(new
    GraspStudio::GenericGraspPlanner(grasps,
    qualityMeasure, approach, 0.2, true));
```

Plan a Grasp

Grasps can now be planned by calling the plan method of the grasp planner object. Depending on the parameters specified on construction, the grasp planner tries to create the specified number of grasps while respecting the timeout in milliseconds.

```
// Search one grasp with a timeout of 1000
    milliseconds
int graspsPlanned = planner->plan(1,1000);

// retrieve the pose of the grasp
if (graspsPlanned==1)
{
    // mGrasp is the pose of the grasp applied to the
        global object pose, resulting in the global TCP
        pose which is related to the grasp
    Eigen::Matrix4f mGrasp = grasps->getGrasp(0)->
        getTcpPoseGlobal(object->getGlobalPose());
    // now the eef can be set to a position so that it
        's TCP is at mPose
    eefCloned->setGlobalPoseForRobotNode(eefCloned->
        getEndEffector("Left Hand")->getTcp(),mGrasp);
}


// the last computed quality can be retrieved with
float qual = qualityMeasure->getGraspQuality();
bool isFC = qualityMeasure->isGraspForceClosure();


// The contacts information can be retrieved, by
    closing the end effector
// Additionally the visualization of eefCloned will
    show the closed fingers
std::vector< VirtualRobot::EndEffector::ContactInfo >
    contacts = eefCloned->getEndEffector("Left Hand")->
    closeActors(object);
```

Store a Set of Grasps

Grasps can be easily managed by ManipulationObjects, as shown in the following example. A new object is created and cloned instances of the visualization and the collision model are passed to the constructor. Then the set of planned grasps is added and the object is stored to an XML file. The XML file will contain the filename of the visualization model and all grasping information.

```
// create a new object, pass cloned visualizations and
    collision models to it.
VirtualRobot::ManipulationObjectPtr newObject(new
   VirtualRobot::ManipulationObject(object->getName(),
   object->getVisualization()->clone(),object->
   getCollisionModel()->clone()));

// append the set of planned grasps
newObject->addGraspSet(grasps);

// save to XML file
try
{
    ObjectIO::saveManipulationObject(newObject, "
       ObjectWithGraspSet.xml");
}
catch (VirtualRobotException &e)
{
    cout << "␣ERROR␣while␣saving␣object" << endl;
    cout << e.what();
}content...
```

Grasp Stability

Simox constructs 6D wrenches from the contact information to represent the contact force and torque. Second, a quality measure is obtained from the analysis of the convex hull of all contact wrenches. The resulting grasp is considered to be force-closure by Simox, if the convex hull contains the origin of the wrench space as an interior point. Third, the minimum distance between the origin of the wrench space and the convex hull surface demonstrates the ability of the grasp to compensate for external disturbances; hence, it is used by Simox as the grasp quality metric

Issues

Summarizing Grasp Studio

2.4.5 Summarizing Simox Archtecture

Based on the robot definitions, any end effector can be decoupled from the model and considered for grasp planning. The Grasp Center Point (GCP) of an end effector defines the favorite grasping position and an ap- proach direction. **A generic grasp planner consists of a module for creating approaching motions and a second module for evaluating the grasp quality.** Both components are exchangeable by custom implementations of the pro- vided interfaces. Further, custom grasp planners, which do not rely on the presented planning loop, are explicitly supported

instead of being randomly sampled as done by the generic grasp planner in Simox.