Readme for Real Time Crcl Trajectory Controller in ROS

Michaloski, John L. (Fed)

7/21/2016 4:32:00 PM

NistControllerReadme.docx

This document presentsa Robot Operating System (ROS) package for a trajectory motion and gripper open/close control that accepts Canonical Robot Control Language (CRCL) commands and reports robot status using ROS subscribe and advertise communication topics.

This implementation provides a simulation that is displayed in RVIZ yet differs from other ROS trajectory packages, e.g., moveit, in that it does not use the trajectory or kinematic functionality of moveit. It does use the Unified Robot Description Format (URDF). The URDF robot description is read using a C++ developed by David Lu that also supports the Kinematics and Dynamics Library (KDL) from Orocos to solve the forward and inverse kinematics of a robot represented in URDF. In addition, the ikfast standalone solution for the Fanuc LR Mate 200 iD robot are available to perform forward and inverse kinematics.

The version information for the Real Time Crcl Trajectory Controller is:

* ROS indigo
* OS - Ubuntu 12.04 (64-bit)
* Package versions in Appendix I

# Canonical Robot Control Language (CRCL) Background

Canonical robot command language (CRCL) is part of the robot research at NIST. CRCL is a messaging language for controlling a robot. CRCL commands are executed by a low-level device robot controller. The usual source of CRCL commands is a plan/program execution system. CRCL is intended for use with devices typically described as industrial robots and for other automated positioning devices such as automated guided vehicles (AGVs). An AGV with a robotic arm attached may be regarded as a single robot responding to a single stream of CRCL commands or as two robots responding to two separate streams of CRCL commands.

Although CRCL is not a programming language, the commands are in the context of a session consisting of getting ready for activity, performing activities, and becoming quiescent. CRCL commands may be collected in files for testing purposes, but executing such files (by giving the commands in the order they occur in the file) is not be the normal operating mode of a robot. Because robots operate in uncertain and changing environment, the reliance on sensors to adjust for such disturbances makes canned scripts ineffective under real conditions.

CRCL models a status message from a low-level robot controller. Status includes the position and orientation (Poses) that are the subject of CRCL commands. If any joint status reporting is done, it is assumed that the system sending canonical commands and the system executing them both know the kinematics of the robot and have the same numbering system for the joints, starting with 1. The two systems also have the same understanding of where the zero point is and which direction is positive for each joint. Status items for joints must be configured using a CRCl ConfigureJointReports command. For each joint for which anything is to be reported, ConfigureJointReports specifies:

* whether joint position should be reported
* whether joint torque or force should be reported
* whether joint velocity should be reported

During a CRCL session, until a ConfigureJointReports command has been executed that sets the reporting status for a joint, default joint status is reported for that joint. The ConfigureJointReports command may be used more than once during a session to change joint status reporting.

# Software Architecture

The controller handles a Fanuc LRMate 200 iD and a robotiq two finger gripper, as shownin the figure below:

# 

The robot and gripper are modeled in ROS URDF shown below:



The following diagram describes the ROS topic communication between modules in the contol system.



The Robot model combines the Fanuc LR Mate 200 id with the robotiq 2 finger gripper. The controller advertised updates to the /nist\_controller/robot/joint\_states which is read by the joint\_state\_publisher package. This communication is enabled in the launch file by the following snippet:

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher">

<param name="/use\_gui" value="true"/>

<rosparam param="/source\_list">[nist\_controller/robot/joint\_states]</rosparam>

</node>

The architecture also supports CRCL command input and status reporting via a CRCL package. NIST CRCL package accepts CRCL commands and reports status to connected ip and socket ports and then published commands using the topic /crcl\_command and subsribes to topic /crcl\_status to receive feedback. This CRCL communication is enabled in the launch file by the following snippet:

<node name="nistcrcl" pkg="nistcrcl" type="nistcrcl" respawn="false" output="screen" >

<param name="crclip" value="127.0.0.1"/>

<param name="crclport" value="64444"/>

</node>

This snippet loads the ROS node "nistcrcl" with the parameters specified as ip (i.e., crclip) equal to "127.0.0.1" and the port (i.e., crclport) equal to 64444.

# Robot Kinematic Chain

The robot path is specified I terms of a "kinematic chain " made up of a series of homogeneous matrix transforms relation the manipulator to the task.



Figure Position Equation

This kinematic chain is evaluated many times a second, each time providing a new set of joint angles positions for the manipulator to follow. This kinematic chain will execute a function each sample period that returns a 4x4 Homogeneou Transform that defines the position and orientation of that element. The Trajectory Generator will use the values in the kinematic chain to solve the kinematics for a robot. Sensor integration is accomplished with the same mechanism; transforms are determined in real time based on sensor input instead of by a static transform.

The MotionEquation class is responsible for assembling a kinematic chain. It uses a static formula to build a kinematics chain, and then each slot uses a callback to a boost statically bound function pointer. The default function pointer returns an identity function.

A Kinematic Chain is assembled suing the make\_equation method. A chain is constructed providing a name, kinematic solver for the robot and then a a series of MotionEquation enums specify the equation layout .which form an equation with a left hand side, and a right hand side. The enumeration EQUALS divides the equation into the left and right hand sides.

RCS::Pose Base(Quaternion(0, 0, 0, 1), Vector3(0, 0, 0));

RCS::Pose Robot(Quaternion(0, 0, 0, 1), Vector3(0, 0, 0));

RCS::Pose Gripper(Quaternion(0, 0, 0, 1), Vector3(.120, 0, 0));

RCS::Pose Table(Quaternion(0, 0, 0, 1), Vector3(0, 0, 0));

RCS::Pose GoalPose(Quaternion ( Vector3(0, 1, 0), 1.57), Vector3(0.25, -.45, 0.35));

// Works

// RCS::Pose GoalPose(Quaternion (0, 0, 0, 1), Vector3(0.465, 0, 0.695));

// RCS::Pose GoalPose(Quaternion ( Vector3(0, 1, 0), 1.57), Vector3(0.465, 0, 0.695));

//RCS::Pose GoalPose(Quaternion (0, 0, 0, 1), Vector3(0.465, 0, .335));

KinematicChain::MotionEquation chain;

chain.make\_equation("Test", kin,

KinematicChain::MotionEquation::BASE,

KinematicChain::MotionEquation::ROBOT,

KinematicChain::MotionEquation::TOOL,

KinematicChain::MotionEquation::EQUALS,

KinematicChain::MotionEquation::TABLE,

KinematicChain::MotionEquation::GOAL,

KinematicChain::MotionEquation::DONE

);

//chain.SetPoseCallback(KinematicChain::MotionEquation::GOAL, boost::bind(&KinematicChain::MotionEquation::GetPose, &chain, \_1));

chain.SetPose( KinematicChain::MotionEquation::GOAL, GoalPose);

chain.SetPoseCallback(KinematicChain::MotionEquation::GOAL, boost::bind(&KinematicChain::MotionEquation::GetPose, &chain, \_1));

chain.SetPose( KinematicChain::MotionEquation::TOOL, Gripper);

std::vector<double> joints = chain. Solve();

## Robot Gripper

In general, an end effector is the device at the end of a robotic arm, meant to interact with the environment (Wikipedia, 2016). The exact nature of this device depends on the application of the robot. We are concerned with grasping objects and placing the objects somewhere else. This can be done with a vacuum gripper, but we are interested in the case of using grippers (with 2 fingers) to achieve object manipulation (grasping and releasing).

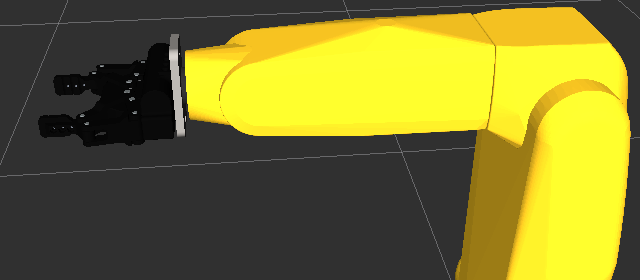
If you use a gripper, you must multiply the inverse of gripper pose (offset) against the pose to get the goal pose for the robot arm. 

Figure 2 Fanuc LR Mate 200iD With Robotiq 2 Finger Gripper

Robotiq’s 2-Finger Adaptive Robot Gripper is modeled as the gripper since a ROS URDF description existed for its kinematics and a CAD model existed to described it visually. Of concern, is determining the gripper offset, that is what is the length offsets of the x,y,z axes of the gripper when it is attached to the robot. To understand the xyz gripper offset, the URDF model describes link6 as having the xaxis point straight ahead, the yaxis points to the side and the z axis points up. Figure 3 shows the positioning of the Link 6 axis and the relationship to the gripper.

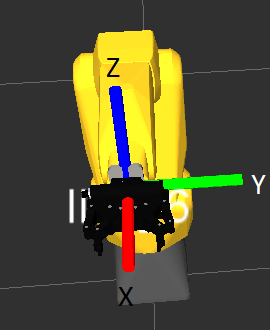


Figure Fanuc Robot Arm Link 6 Axes

So, in the URDF scenario, the y axis is immaterial – the y axis determines the gripper opening, not offsets. While Figure 2 shows the offsets of concern the x and z axis. In our case the x axis describes the length of the gripper. The robotiq description can be found at <http://robotiq.com/products/adaptive-robot-gripper/> gives tshe length of the gripper as 140 mm, which we use as the x translation offset. Notices that the gripper up/down position changes, and this corresponds to a change in the z axis. Since this offset is the negative z direction (down), we will show later how -0.017 meters was determined to be the Z offset.

A special CRCL command was added to allow a kinematic ring with gripper offset to be defined as something other than the identity matrix. Below, the RCS (real time control system) canonical command is given that describes the gripper offset pose as a constructor combination of an identity quaternion and translation offset..

void AddGripperOffset(){

RCS::CanonCmd cmd;

cmd.crclcommandnum = crclcommandnum++;

cmd.crclcommand = CanonCmdType::CANON\_SET\_GRIPPER\_POSE;

cmd.finalpose = Conversion::RcsPose2GeomMsgPose(

RCS::Pose(tf::Quaternion(0.0, 0.0, 0.0, 1.0),

tf::Vector3(0.140, 0.0, -0.017) )); // -0.01156)));

RCS::Cnc.crclcmds.AddMsgQueue(cmd);

}

When processed by the control system the gripper offset commands sets the kinematic pose component for the gripper and its inverse.

else if (\_newcc.crclcommand == CanonCmdType::CANON\_SET\_GRIPPER\_POSE) {

gripperPose = Conversion::GeomMsgPose2RcsPose(\_newcc.finalpose);

invGripperPose = gripperPose.inverse();

Then any Cartesian motion that has a position and orientation to describe the motion changes the final point destination by postmultiplying the gripper inverse pose against the pose to determine the final robot pose.

RCS::Pose goalpose = finalpose \* Cnc.invGripperPose ;

Because we didn't actually have a robotiq 2-finger gripper, we couldn't just measure the z axis offset. Instead, a simple forward kinematic solution from the robotiq base joint to the final joint was calculated to give the Z axis. The hard coded solution will be shown to achieve the forward kinematic position, even though the links and axis of rotation and position and rotation transform from the parent link can be determined from the URDF which was already parsed.

RCS::Pose ComputeGripperOffset() {

AllM.push\_back(ComputeUrdfTransform(0.0, Eigen::Vector3d(1, 0, 0), Eigen::Vector3d(.0085 ,0 ,-.0041), Eigen::Vector3d(0, 0, 0)));

AllM.push\_back(ComputeUrdfTransform(0.0, Eigen::Vector3d(1, 0, 0), Eigen::Vector3d(.04191, -.0306, 0), Eigen::Vector3d(1.5707, - 1.5707, 0)));

AllM.push\_back(ComputeUrdfTransform(0.0, Eigen::Vector3d(0, -1, 0), Eigen::Vector3d(0, .00508, .03134), Eigen::Vector3d(3.1415, 0, 0)));

AllM.push\_back(ComputeUrdfTransform(0.0, Eigen::Vector3d(-1, 0, 0), Eigen::Vector3d(.04843 ,- .0127, 0), Eigen::Vector3d(-1.5707, - 1.5707, 0)));

AllM.push\_back(ComputeUrdfTransform(0.0, Eigen::Vector3d(0, -1, 0), Eigen::Vector3d(0 ,.04196, - .0388), Eigen::Vector3d(0, 0, 0)));

RCS::Pose pose = ComputeFk();

LOG\_DEBUG << "Gripper Offset Pose " << RCS::DumpPoseSimple(pose).c\_str();

}

This code manually loads the 5 robotiq URDF joint information, and compute the Forward Kinematics (with zero joint angles) to determine the x and z offset. The x axis offset only goes to the final knuckle and not all the way down the gripper pinchers.

# RVIZ Visualization

The use of Rviz in simulation and visualization of the robot trajectory behavior is an important element in deploying the CRCL controlled robot. The CRCL includes Cartesian, joint and gripper control that is handled by the controller.

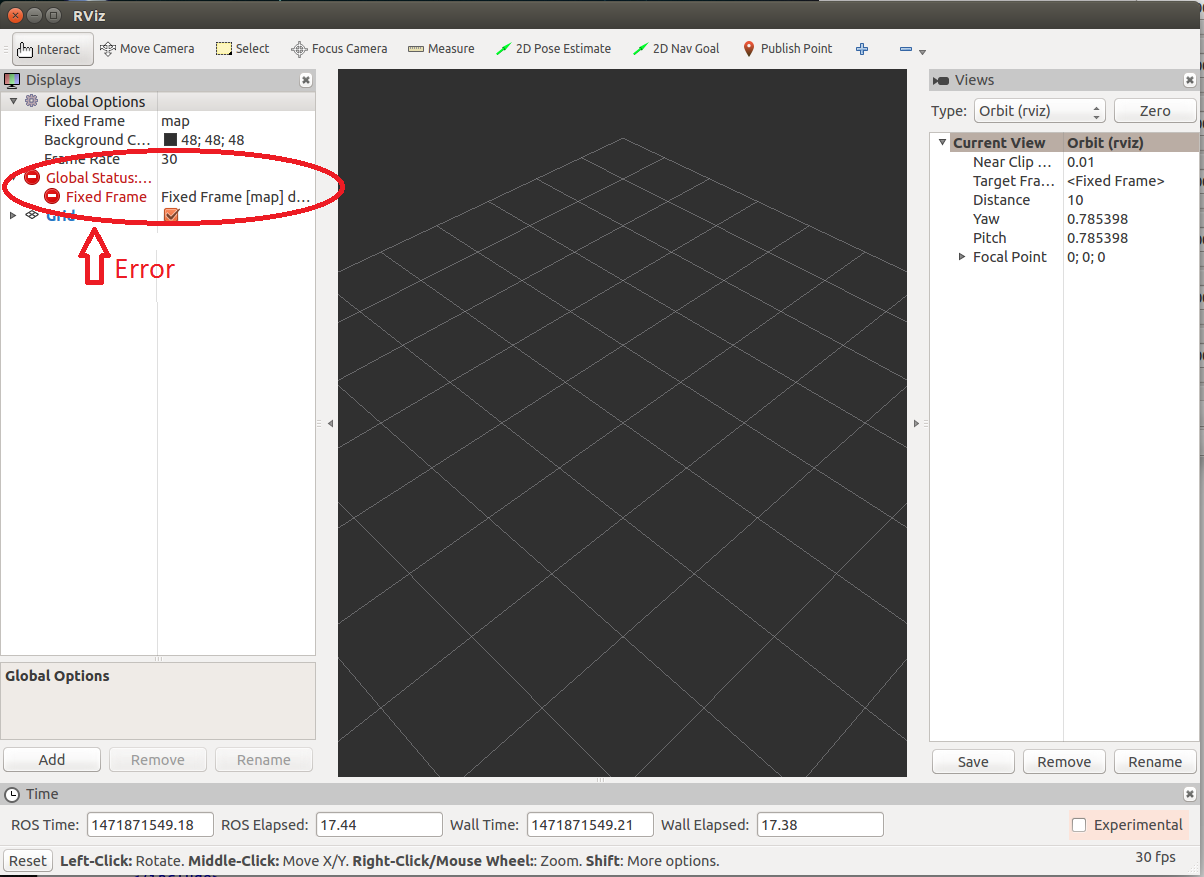
Rviz visualization is a nice robot visualization tool, but many of the elements are only explained in the context of implementations based on the Willow Garage PR2 robot. Thus, many of the tutorials although helpful, are bundled with other packages making it monolithic and often feel like coding with a heap of spaghetti. However, the source code is available and noodling around in the source code and by searching far and wide across the Internet, pearls of ROS programming can be found and where integrated into the package, and are hopefully understandable in this documentation. This section will attempt to explain how to use Rviz (without moveit planning and obstacle avoidance) to visualize a robot scene. True, eventually you will probably have to use moveit planning and obstacle avoidance, but one sip from a fire hose at a time.

The easies first step is to use roslaunch, in which you load a robot description and a "stripped down" version of Rviz.

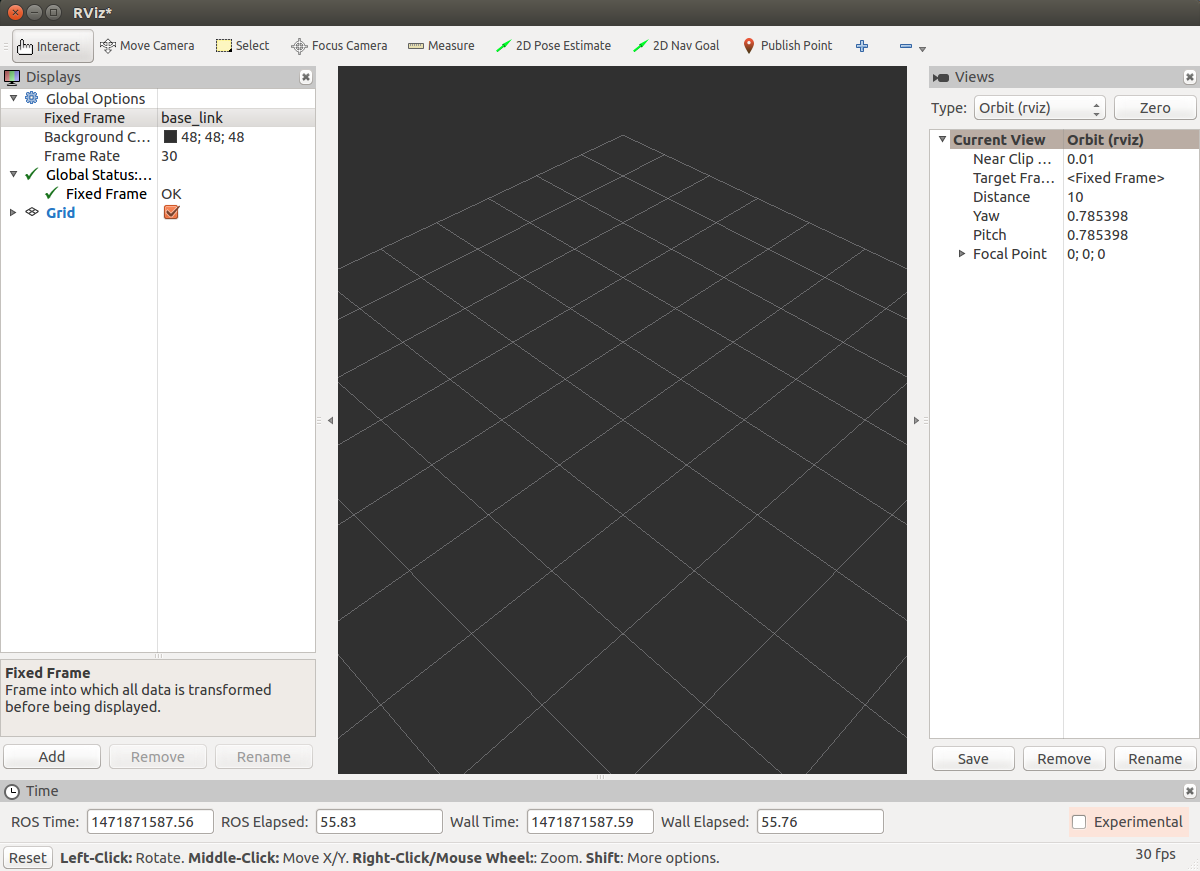
<param name="robot\_description" command="$(find xacro)/xacro.py $(find fanuc\_lrmate200id\_support)/urdf/lrmate200id.xacro" />

<node name="rviz" pkg="rviz" type="rviz">

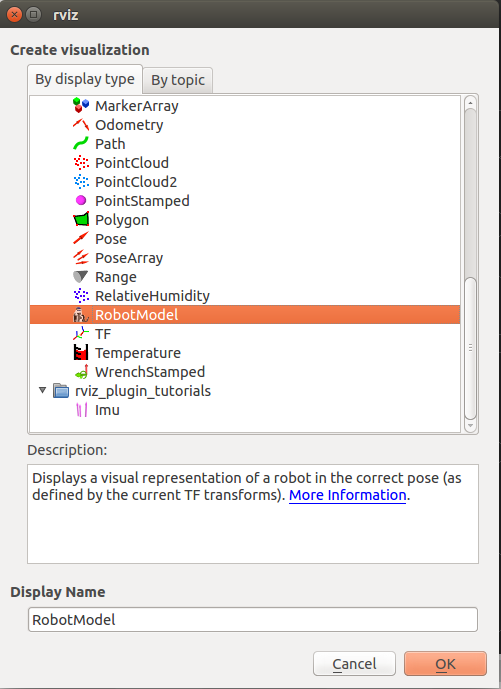
When you do this, you will eventually see an RVIZ screen appear with the error condition of "Global Status".



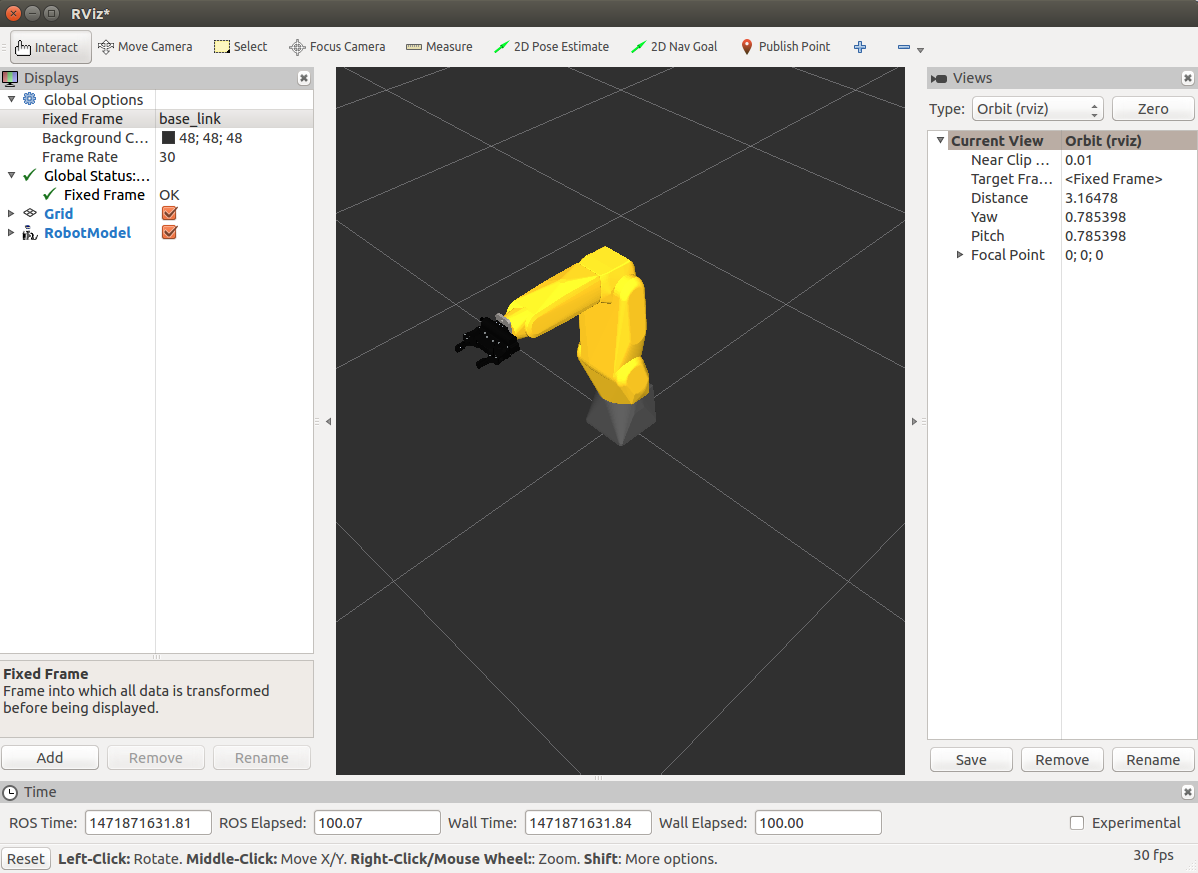
To rectify this error, click on the Fixed Frame text box (and possibly the base\_link will appear in a combo box which you can select) or type in "base\_link" or whatever is the base link in your URDF robot description. Below the error message disappears when base\_link is entered.



Now, the problem is that no robot is visible. Obviously, not a good situation. To rectify this problem, click the [ADD} button above time in the lower left hand corner, and when the Create Visualization dialog box appears, select "Robot Model", as shown below:



Then, the robot that is described in the robot\_description ROS parameter will appear in the RVIZ visualization, as shown below. The robot shown below is a Fanuc LR Mate 200 Id with a 2 finger robotiq gripper attached.



Next, moving the robot is important. Two packages are useful in moving the robot: robot\_state\_publisher and joint\_state\_publisher.

* robot\_state\_publisher allows you to publish the state of a robot to tf. Once the state gets published, it is available to all components in the system that also use tf. The package takes the joint angles of the robot as input and publishes the 3D poses of the robot links, using a kinematic tree model of the robot.
* joint\_state\_publisher publishes sensor\_msgs/JointState messages for a robot. The package reads the robot\_description parameter, finds all of the non-fixed joints and publishes a JointState message with all those joints defined. joint\_state\_publisher is used in conjunction with the robot\_state\_publisher node to also publish transforms for all joint states.

Of importance is the ROS parameter :ource\_list", which is a list of topics that the "joint\_state\_publisher" node listens for sensor\_msgs/JointState messages. Below, the "joint\_state\_publisher" node source list contains "nist\_controller/robot/joint\_states" topic which is listened to for new joint position to update the published joint\_state. In this manner, the Real Time Crcl Trajectory Controller published either arm or gripper joints to the "nist\_controller/robot/joint\_states" topic, which the " joint\_state\_publisher" node listens to and republished on the "joint\_states" topic that RVIZ is listening to for joint updates.

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="state\_publisher" output="screen" />

<!-- We do not have a robot connected, so publish fake joint states -->

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher">

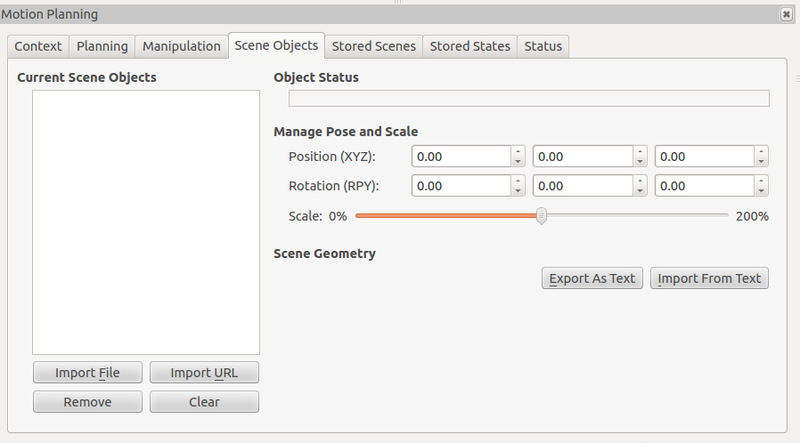
<param name="/use\_gui" value="true"/>

<rosparam param="/source\_list">[nist\_controller/robot/joint\_states]</rosparam>

</node>

# Manual Inserting of an STL File containing a scene object in RViz

A scene of objects for gripper manipulation by the Crcl Robot Controller in RVIZ must be built. Each object is imported into RVIZ as a Stereolithography Language (STL) file. To import an object, a 3D STL file is imported with the Scene Objects-Import File button. The STL format should be binary, not plain text. (?)



RVIZ is independent of moveit, so to get objects registered in the moveit Planning Scene, collision object programmatically to moveit (See http://wiki.ros.org/motion\_planning\_environment/Tutorials/Adding%20known%20objects%20to%20the%20collision%20environment) .

# RVIZ SCENE CREATION

This section covers the addition of objects to an RVIZ scene programmatically using C++. The scene creation uses the rviz-visual-tools ROS package, details found at <https://github.com/davetcoleman/rviz_visual_tools/blob/kinetic-devel/README.md>. This Web site includes a tutorial on rviz-visual-tools using RVIZ which gives a good background to the ROS package functionality.

For the Fanuc LR Mate 200iD, the addition of two objects will be illustrated to show how to use the rviz visual tools. Rviz visual tools uses the marker\_array topic to publish object into rviz. Typically, one might use the moveit “collision objects” to display scene objects in Rviz.

Two objects will be added to the Rviz scene, a medium gear and a gear holder tray. These scene objects were created in a CAD design system and have been produced by an 3D printing device. 3D printing devices use STL, so the STL files from these objects were imported and displayed Rviz using the displayMesh rviz-visual-tools method.

In addition, the method publishWall was developed that is based on rviz-visual-tools, to display a wall in Rviz.

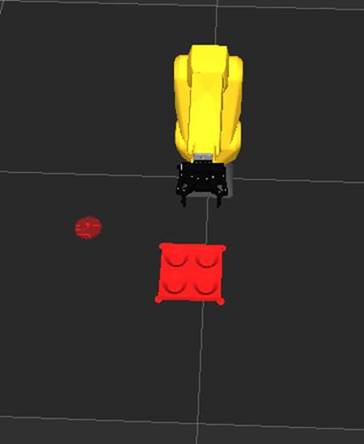


Figure Gear and Gear Holder Objects Displayed in Rviz Scene

On the Ubuntu 12.4 platform that ROS was running indigo version , rviz-visual-tools was required to be installed. Using instructions from <https://github.com/davetcoleman/rviz_visual_tools>, the following command line performed the installation:

sudo apt-get install ros-indigo-rviz-visual-tools

The initial test code was developed as C++ functions inside of a CPP implementation file. The rviz-visual-tools package uses Eigen math library to specify positions and orientation, specifically, representing object locations as Homogeneous Transform matrices or Eigen::Affine3. Documentation for Eigen Affine representation can be found here: https://eigen.tuxfamily.org/dox-devel/group\_\_TutorialGeometry.html

#include <rviz\_visual\_tools/rviz\_visual\_tools.h>

using namespace rviz\_visual\_tools;

rviz\_visual\_tools::RvizVisualToolsPtr visual\_tools;

void InitSceneObject() {

visual\_tools = boost::shared\_ptr<RvizVisualTools>(new RvizVisualTools("base\_link", "/visualization\_marker\_array"));

visual\_tools->deleteAllMarkers();

visual\_tools->enableBatchPublishing();

}

void SetupSceneObject()

{

Eigen::Affine3d pose= Eigen::Affine3d::Identity()\*Eigen::Translation3d(0.5, 0, 0.0);

Eigen::Affine3d pose4= Eigen::Affine3d::Identity()\*Eigen::Translation3d(0.25, -.45, 0.0);

bool b;

if (!(b=visual\_tools->publishMesh(pose, // or const geometry\_msgs::Pose &pose

"file:///usr/local/michalos/nistfanuc\_ws/src/nist\_fanuc/worldmodel/medium\_gear\_holder.stl",

rviz\_visual\_tools::RED, // const colors &color = CLEAR,

0.035, // double scale = 1,

"", // const std::string &ns = "mesh",

1))) { // const std::size\_t &id = 0))

std::cout << "SetupSceneObject() Failed\n";

}

visual\_tools->triggerBatchPublish();

#if 1

visual\_tools->publishMesh(pose4, // or const geometry\_msgs::Pose &pose

"file:///usr/local/michalos/nistfanuc\_ws/src/nist\_fanuc/worldmodel/medium\_gear.stl",

rviz\_visual\_tools::RED, // const colors &color = CLEAR,

0.035, // double scale = 1,

"", // const std::string &ns = "mesh",

2);

visual\_tools->triggerBatchPublish();

#endif

}

First, you need to modify ROS package.xml and CMakeList.txt to include references to rviz\_visual\_tools. The catkin build will fail if the rviz\_visual\_tools has not been installed. Without this package, include files, and the rviz\_visual\_tools library cannot be found.

Assuming the rviz\_visual\_tools has been installed, then you need to include its header file to reference its functionality:

#include <rviz\_visual\_tools/rviz\_visual\_tools.h>

Next, you need to declare a RvizVisualToolsPtr from the namespace rviz\_visual\_tools to your code (either in your class or as a global variable):

// For visualizing things in rviz

rviz\_visual\_tools::RvizVisualToolsPtr visual\_tools\_;

Please note, DO NOT instantiate the declaration with an instance, because it will attempt to do so before you have called ros\_init in you main file, and will cause an error message.

The function InitSceneObject was defined to instantiate visual\_tools with a shared\_ptr instance of RvizVisualTools, and to do some preliminary initialization (i.e., deleteAllMarkers and enableBatchPublishing) :

void InitSceneObject() {

visual\_tools = boost::shared\_ptr<RvizVisualTools>(new RvizVisualTools("base\_link", "/visualization\_marker\_array"));

visual\_tools->deleteAllMarkers();

visual\_tools->enableBatchPublishing();

}

Note, change the first parameter to the name of your robot's base frame (i.e., “base\_link), and the second parameter to whatever name you'd like to use for the corresponding Rviz marker ROS topic (can see it rviz under marker array).

Now we create some scene objects using the function SetupSceneObject. In the following code we create two poses, one at xyz (0.5, 0, 0) with no orientation, and the other at pose (0.25, -.45, 0). Then the gear and the gear holder STL meshes are displayed by the publishMesh method. The color chosen was rviz\_visual\_tools::RED and each ID must be unique or only the last item with the id will be displayed. There is a namespace which was declared to be a "mesh", and a scaling factor (i.e., 0.35) when displaying the STL file. Unfortunately, other STL files were represented with millimeters, but by trial and error 0.35 seems to work.

void SetupSceneObject()

{

Eigen::Affine3d pose1= Eigen::Affine3d::Identity()\*Eigen::Translation3d(0.5,0,0);

Eigen::Affine3d pose2= Eigen::Affine3d::Identity()\*Eigen::Translation3d(0.25,-.45,0);

visual\_tools->publishMesh(pose1, "file:///usr/local/michalos/nistfanuc\_ws/src/nist\_fanuc/worldmodel/medium\_gear.stl",

rviz\_visual\_tools::RED,

0.035,

"mesh", // mesh namespace",

1); // id must be unique, 0 used package to assign id

visual\_tools->triggerBatchPublish();

visual\_tools->publishMesh(pose2, "file:///usr/local/michalos/nistfanuc\_ws/src/nist\_fanuc/worldmodel/medium\_gear.stl",

rviz\_visual\_tools::RED,

0.035,

"", // no namespace",

2);

visual\_tools->triggerBatchPublish();

Note, the use of an STL file mesh needs to be specified as a URI file (with leading file://) for Rviz to understand that it is a file and where the STL file can be located. Errors ini the ROS console will appear if you incorrectly specify the file to Rviz.

To test, use roslaunch to start the simple.launch file. This launch file will start rosmaster, rviz and set the robot\_description ROS parameter. In Rviz, you need to make sure the Marker Array is established as an Rviz module. To do this, press the 'Add' button at the bottom right and select Marker Array module. Note, the marker array topic MUST be the same as the topic you specified in the RvizVisualTools constructor or Rviz will not listen to any marker arrays published by rviz\_visual\_tools.

## TRAJ Trajectory Planning Algorithms

Trajectory planning functions abbreviations:

* CV means constant velocity, CA means constant acceleration,
* CJ means constant jerk.

The Go Motion trajectory planning algorithms are based on smooth velocity profiling with bounded speed, acceleration and jerk, called "constant jerk" or "S-curve" velocity profiling. This gives smoother control than "trapezoidal" velocity profiling, which transitions instantaneously between acceleration and no acceleration and incurs spikes in unbounded jerk.

Constant-jerk (CJ) profiling is shown in Figure 1, a plot of the speed versus time. There are 7 phases to the motion. Phase 1 is a jerk phase, where the acceleration varies smoothly from 0 at time 0 to \a a1 at time \a t1 following the jerk (change in acceleration per unit time) \a j0. Phase 2 is an acceleration phase, with constant acceleration \a a1 throughout. Phase 3 is a jerk phase (or de-jerk phase) with constant (negative) jerk slowing down the acceleration from \a a1 to 0. Phase 4 is a constant speed phase at speed \a v3. Phase 5 is a constant-jerk counterpart to phase 3, where the deceleration varies smoothly from 0 to \a -a1. Phase 6 is a constant-acceleration counterpart to phase 2. Phase 7 is a constant-jerk counterpart to phase 1, where the deceleration varies smoothly from \a -a1 to 0 and motion stops.

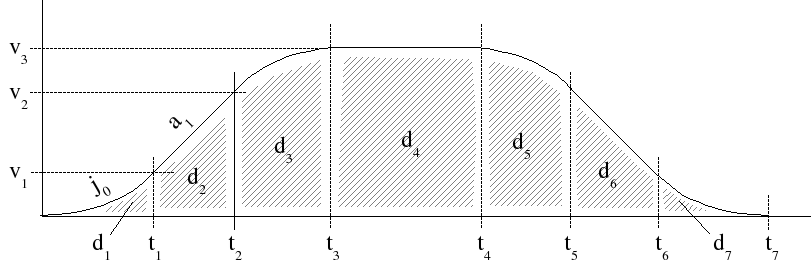
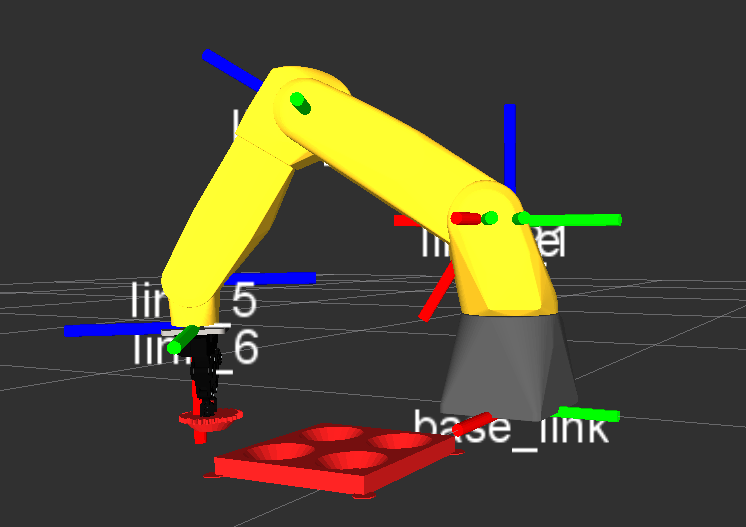


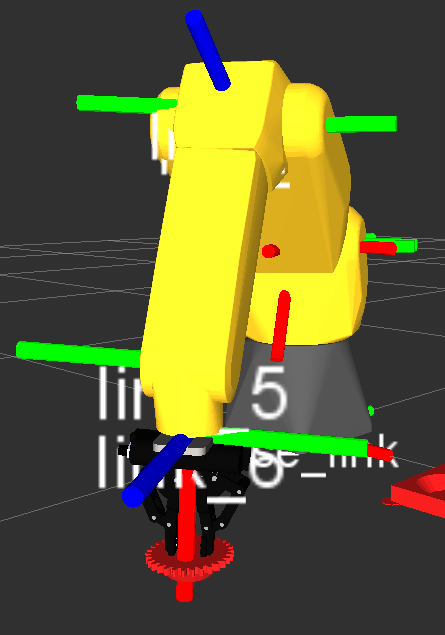
Figure Constant jerk velocity profiling.

## Rviz display of the robot tf (transform) display

In order to get a visualization of the axes for each axis of you robot, rviz can offer this service is you ADD the "TF" module. Assuming you have started Rviz and configured it so that there is a robot description and the Robot Model module has been added to Rviz, you can turn on the axes visualization for the links you desire to visualize them. Below, link\_1 through link\_6 have axis visualization enabled :



Here is another vantage point:

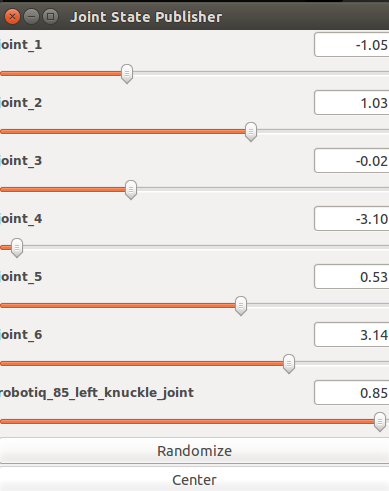


The X axis is indicated in red, the Y axis is indicated in green, and the Z axis is indicated in blue. (<http://wiki.ros.org/rviz/DisplayTypes/TF>). Thus, in the scene above the bolt is located at (.25,-45,0) which is not the centroid of the object.

## Finding the pose of an object in the RVIZ scene

The STL meshes mapping into the Rviz scene use a pose to position the STL object. At this point in time, this Rviz mapping of the object pose is that it is not a centroid or it would be assumed the centroid of the pose to place the bolt object would give the bolt location – and it does not.

Instead trusty joint publisher GUI has sliders to move the joint values around to place the robot with the correct position and orientation to pick up the bolt. Suffice to say that it was not trivial centering the robot over the bolt but can be done. You can get the joint positions on the Joint State Publisher GUI :



Assuming you have started Rviz and configured it so that there is a robot description and the Robot Model module has been added to Rviz, you can read the position and orientation of link\_6 which should place the robot in the correct position to grasp the bolt:

