Readme for Real Time Crcl Trajectory Controller in ROS

Michaloski, John L. (Fed)

7/21/2016 4:32:00 PM

NistControllerReadme.docx

This document presents the robot controller with trajectory motion and gripper open/close control developed for the Robot Operating System (ROS) package 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) and Kinematics and Dynamics Library (KDL) from orocos to soved the forward and inverse kinematics of a robot represented in URDF.

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.

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



Figure Position Equation

This position equation is evaluated many times a second, each time providing a new set of joint angles positions for the manipulator to follow. This type of transform will execute a function each sample period containing an equation to define its values. The Trajectory Generator will use the new values in the position equation. Sensor integration is accomplished in the same manner, new transforms are determined by sensor input instead of by equations.

Sample program:

#include typedef.h"

void cybotask()

{

DevicePtr robot;

TransformPtr base, tool, table, place\_a, place\_b;

bobot = get\_device("FanucLRMate200iD");

base = make\_transform("BASE", constant, p\_vector,

inches, 0.0, 64.75, -0.5, nostore);

tool = make\_transform("TOOL", constant, p\_vector,

millimeters, 0.0, 0.0, 117.4, nostore);

table = make\_transform("TABLE", constant, p\_vector,

millimeters, 1000.0, 1000.0, 700.0,

rpy\_angles, degrees, 0.0, 0.0, 90.0, nostore);

place\_a = make\_transform("PLACE\_A", constant, p\_vector,

millimeters, 150.0, 100.0, 0.0, nostore);

place \_b= make\_transform("PLACE\_B", constant, p\_vector,

millimeters, 150.0, 250.0, 0.0, nostore);

position\_a = make\_position("POSITION A",

base,robot, tool, EQUALS, table, place-a,

TOO, tool);

position\_b = make\_position("POSITION A",

base,robot, tool, EQUALS, table, place-a,

TOO, tool);

set\_segment\_tim(robot, .l5);

set\_translational\_velocity(robot, 100.0);

set rotoational\_velocity(robot, 10.0);

set\_cartesian\_mode(robot);

move(position\_a)

move(position\_b);

}

# 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.

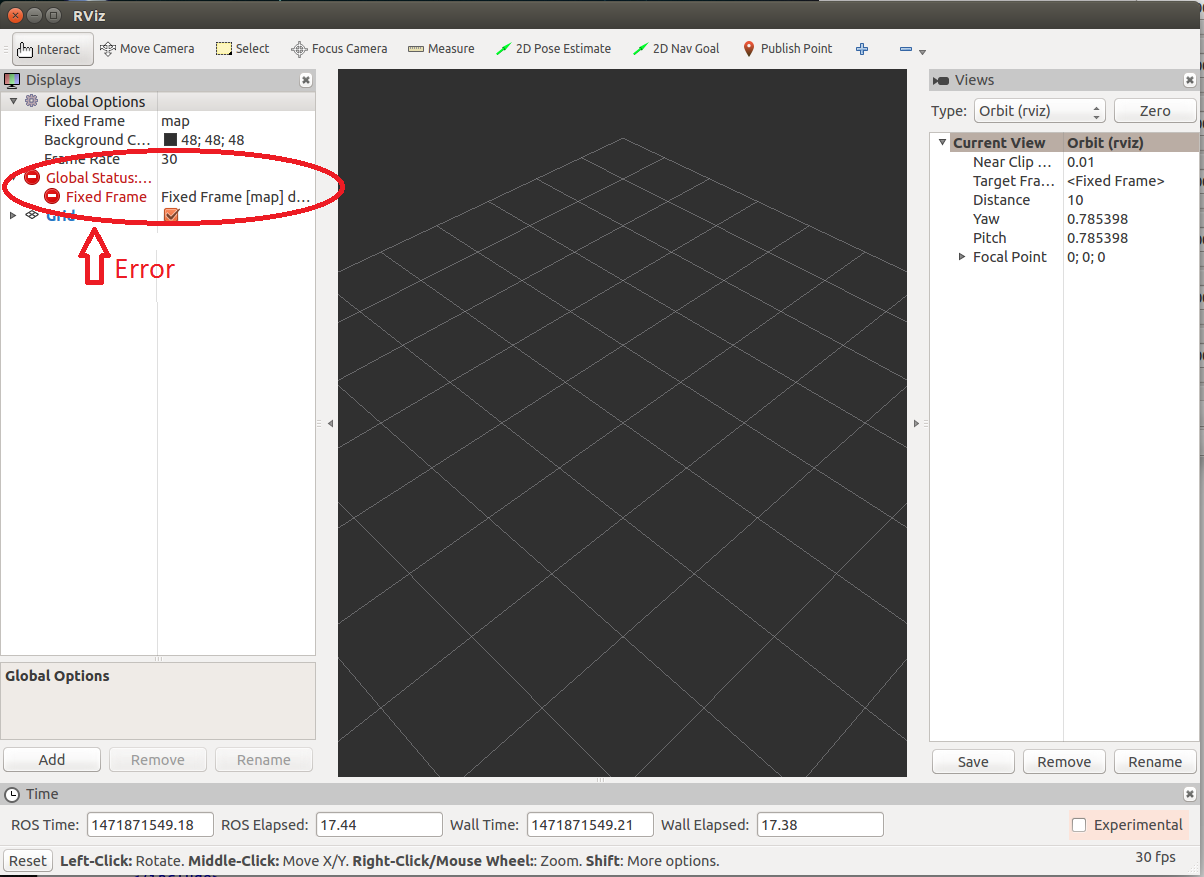
Rviz visualization is a nice robot visualization tool, but many of the lement are only explained in the context of the Willow Garage PR2 robot. Thus, many using the tutorials . However, the source code and noodling around This section will attempat to explain how to use rviz without moveit planning to visualize robot motion.

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

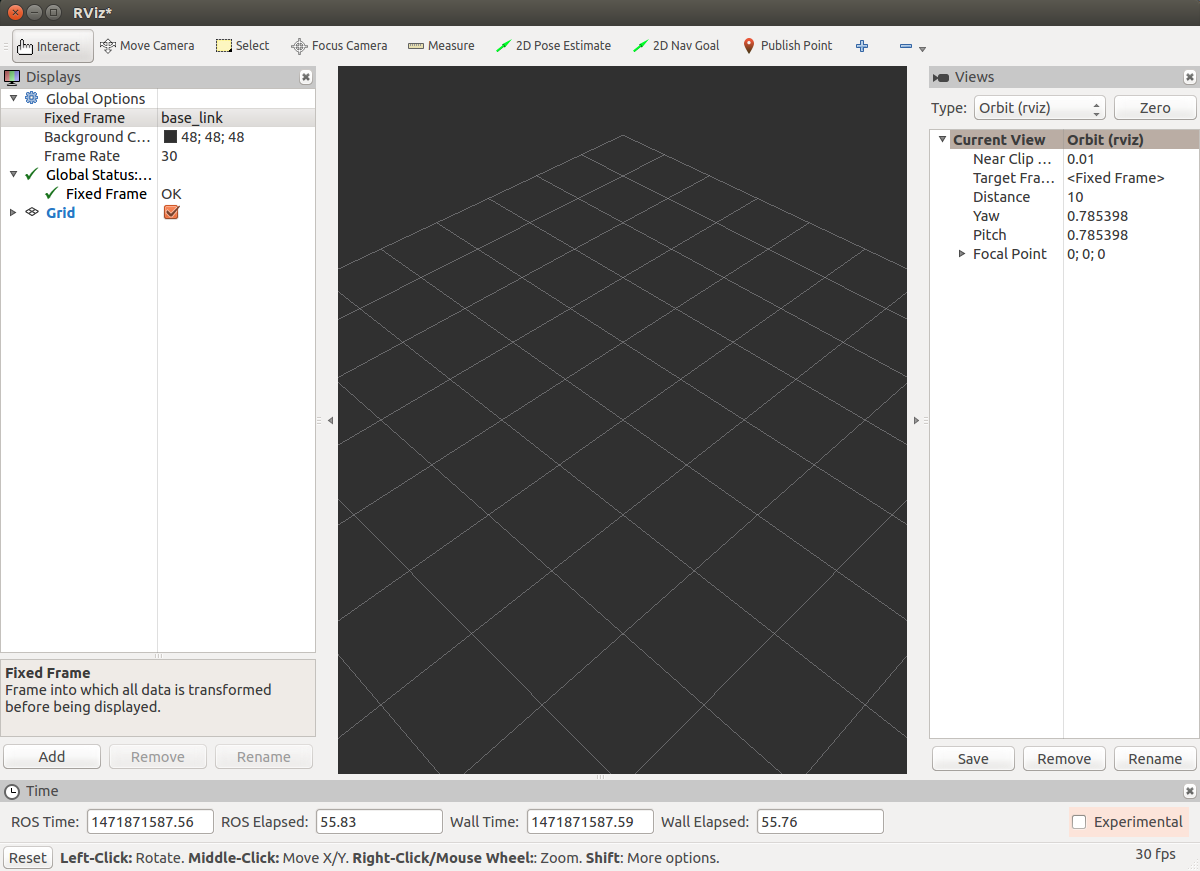
<param name="robot\_description" command="$(find xacro)/xacro.py $(find fanuc\_lrmate200id\_support)/urdf/lrmate200id.xacro" />

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

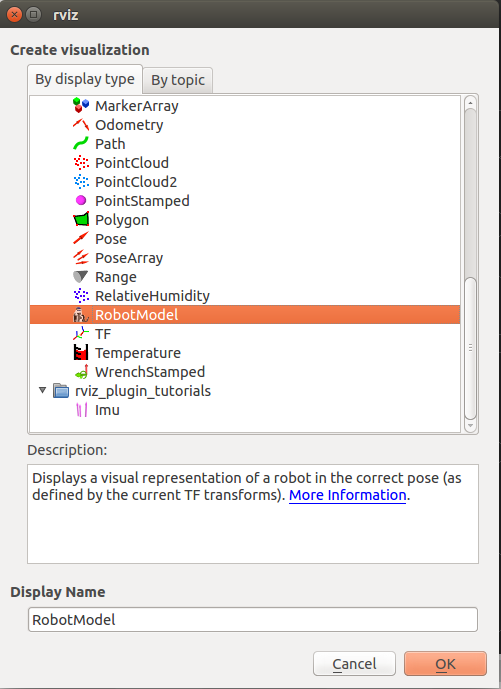
When you do this, you will eventually see an RVIZ screen appear with the error condition of "Global Status".



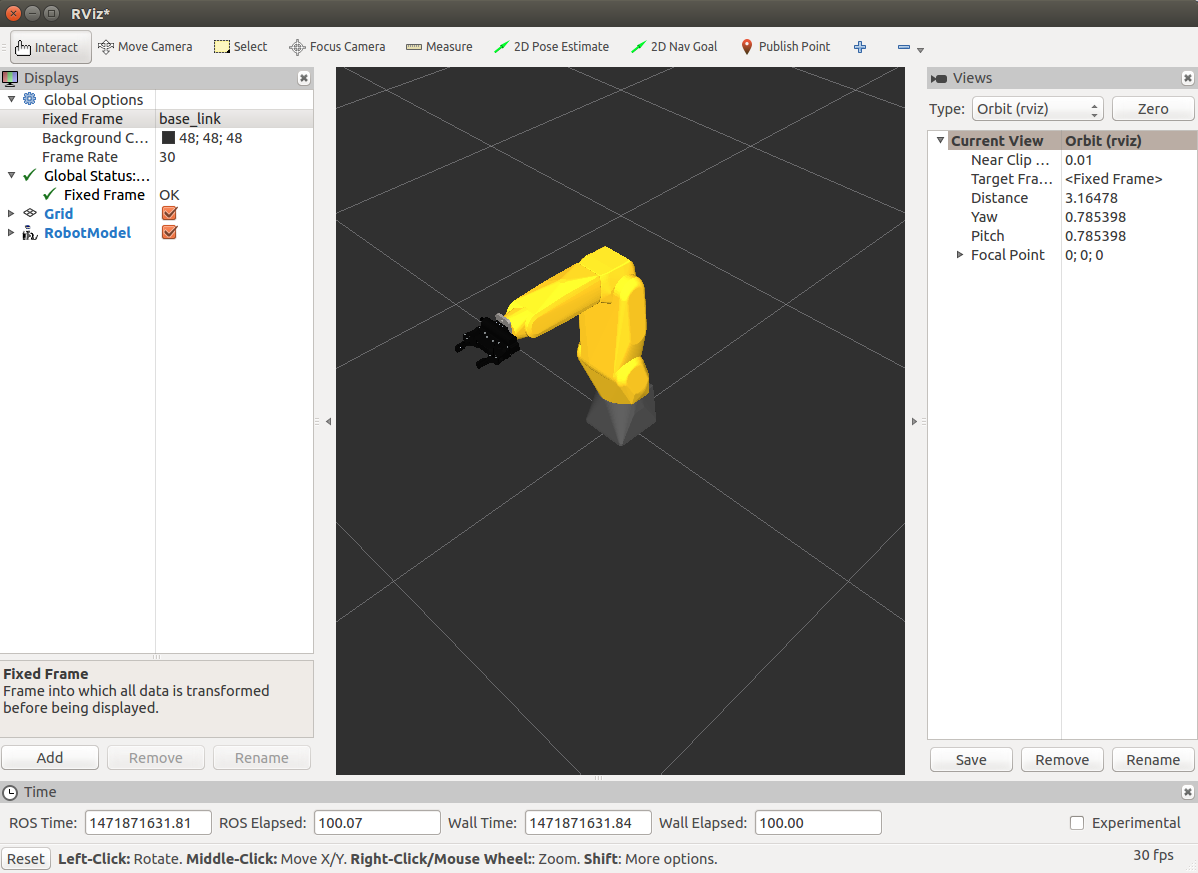
To rectify this error, click on the Fixed Frame text box (and possibly the base\_link will appear in a combo box which you can select) or type in "base\_link" or whatever is the base link in your URDF robot description. Below the error message disappears when base\_link is entered.



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:



The, 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 source\_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>

## 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.

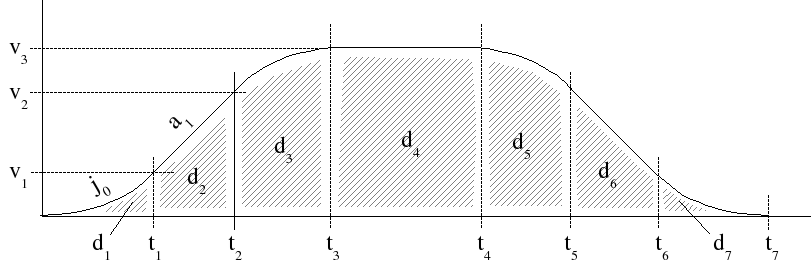
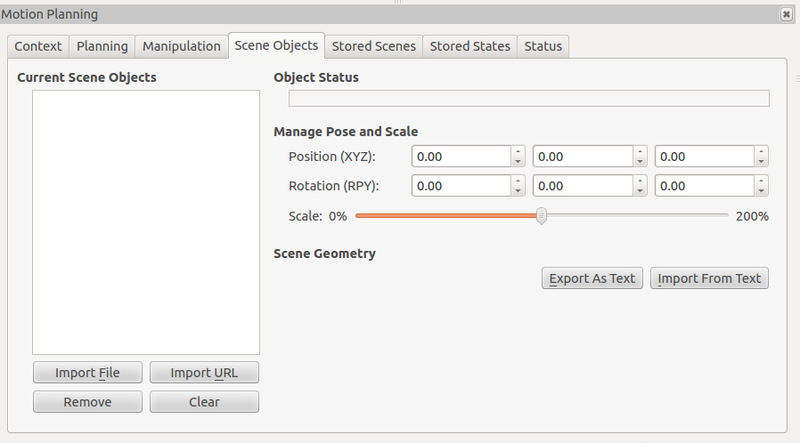


Figure Constant jerk velocity profiling.

**0**

# Inserting STL or 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 how to add objects to the RVIZ scene. From <https://github.com/davetcoleman/rviz_visual_tools/blob/kinetic-devel/README.md> the following discourse on RVIZ is given to help the uninitiated to its functionality.

## Initialize

Add to your includes:

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

Add to your class's member variables:

// For visualizing things in rviz

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

In your class' constructor add:

visual\_tools\_.reset(new rviz\_visual\_tools::RvizVisualTools("base\_frame","/rviz\_visual\_markers"));

Change the first parameter to the name of your robot's base frame, and the second parameter to whatever name you'd like to use for the corresponding Rviz marker ROS topic.

## Tools

Now in your code you can easily debug your code using visual markers in Rviz

Start rviz and create a new marker using the 'Add' button at the bottom right. Choose the marker topic to be the same as the topic you specified in the constructor.

## Example Code

In the following snippet we create a pose at xyz (0.1, 0.1, 0.1) and rotate the pose down 45 degrees along the Y axis. Then we publish the pose as an arrow for visualization in Rviz. Make sure your Rviz fixed frame is the same as the one chosen in the code.

// Create pose

Eigen::Affine3d pose;

pose = Eigen::AngleAxisd(M\_PI/4, Eigen::Vector3d::UnitY()); // rotate along X axis by 45 degrees

pose.translation() = Eigen::Vector3d( 0.1, 0.1, 0.1 ); // translate x,y,z

// Publish arrow vector of pose

ROS\_INFO\_STREAM\_NAMED("test","Publishing Arrow");

visual\_tools\_->publishArrow(pose, rviz\_visual\_tools::RED, rviz\_visual\_tools::LARGE);

## Basic Publishing Functions

See visual\_tools.h for more details and documentation on the following functions:

* publishSphere
* publishSpheres
* publishArrow/publishXArrow
* publishYArrow
* publishZArrow
* publishCuboid
* publishCone
* publishXYPlane
* publishXZPlane
* publishYZPlane
* publishLine
* publishPath
* publishPolygon
* publishBlock
* publishWireframeCuboid
* publishWireframeRectangle
* publishAxis
* publishAxisLabeled
* publishCylinder
* publishMesh
* publishMesh
* publishText
* publishTest

And more...

## Helper Functions

Reset function

* deleteAllMarkers - tells Rviz to clear out all current markers from being displayed.

Batch publish - useful for when many markers need to be published at once to prevent buffer overflow of ROS messages.

* enableBatchPublishing()
* triggerBatchPublish()
* triggerBatchPublishAndDisable()

Conversion functions

* convertPose
* convertPoint32ToPose
* convertPoseToPoint
* convertPoint
* convertPoint32
* convertFromXYZRPY
* convertToXYZRPY

Convenience functions

* generateRandomPose
* generateEmptyPose
* dRand
* fRand
* iRand
* getCenterPoint
* getVectorBetweenPoints

## Available Colors

This package helps you quickly choose colors - feel free to send PRs with more colors as needed

BLACK,

BLUE,

BROWN,

CYAN,

DARK\_GREY,

GREEN,

GREY,

LIME\_GREEN,

MAGENTA,

ORANGE,

PINK,

PURPLE,

RED,

WHITE,

YELLOW,

TRANSLUCENT\_LIGHT,

TRANSLUCENT,

TRANSLUCENT\_DARK,

RAND,

CLEAR,

DEFAULT // i.e. 'do not change default color'

### Available Marker Sizes

XXSMALL,

XSMALL,

SMALL,

REGULAR,

LARGE,

xLARGE,

xxLARGE,

xxxLARGE,

XLARGE,

XXLARGE