Readme for Gomotion ROS Package

12/23/2016 1:24:00 PM

ROSGomotion.docx

## Abstract

This document discusses the Robot Operating System (ROS) package for gomotion robot joint and Cartesian pose trajectory generation. The Gomotion trajectory generation ROS package is a subset of the gomotion controller library (found at https://github.com/frederickproctor/gomotion ), which contains code for a full real-time motion control system for machine tools or robots. Herein the gomotion library for ROS will be known as the gomotion trajectory package as opposed to the gomotion control library from which the code is derived.

The ROS gomotion package accepts robot destinations as either transform poses (from the tf package) or as a position or velocity values within a JointState representation, represented as a C++ std vector of doubles. Internally, go motion has more functionality, but for the ROS package, functionality is limited to joint and Cartesian pose trajectory generation.

Gomotion ROS package for trajectory generation has two fundamental motion types: joint and world.

* Gomotion joint trajectory generation generates a motion plan expressed as joint values for each joint in a mechanism. Starting with the current joint values, gomotion takes destination joint values for which a coordinated or uncoordinated plan is generated. A coordinated joint trajectory has all the mechanism joints arrive at the destination position at the same time. An uncoordinated joint trajectory has the trajectories scaled so that they arrive simultaneously.
* Gomotion world trajectory generation generates a motion plan given a goal destination expressed as a six dimensional "pose", which is combination of 3D position (x,y,z) and an orientation series of rotations around axes x, y, z at the 3D position.

The Gomotion ROS package offers a straightforward means in which to define smooth trajectories based on either serial link (e.g., robot or machine too) dynamic profile (i.e., velocity, acceleration, and jerk) or time. The Gomotion ROS package is not all inclusive, and does not support obstacle detection, the final velocity is always zero, and there is no blending between goal points. TOLERANCES? WAYPOINTS?

## Abbreviations

CA constant acceleration  
CV constant velocity  
CJ constant jerk  
ECP tool's end control point  
KCP kinematic  
TCP tool center point  
EE end effector

## Background

<http://superuser.com/questions/340650/type-math-formulas-in-microsoft-word-the-latex-way>

In the gomotion paradigm, the robot motion assumes a kinematical device, and supplies a geometrical description of motion without regard to the masses or forces involved. Whereas, a dynamical description of motion uses the idea of an “inertial frame,” or a reference frame relative to which motions have distinguished dynamical properties.

Robot, tool, world. We start with some basic geometric motion concepts: position, rotation, vector, orientation.

Positions are vectors that indicate where something is. In the three-dimensional world, three numbers are necessary to indicate position. Internally, Go Motion supports position vectors in several representations: Cartesian, cylindrical and spherical and has functions that convert position from one representation position in another representation. The Cartesian representation uses three numbers **x**, **y** and **z** to represent distances from the origin along three perpendicular axes. The cylindrical representation uses three numbers **r**, **theta** and z to represent radial distance away from the origin, angle around the origin and distance up and down from the origin respectively. The spherical representation uses three numbers **theta**, **phi** and **r** to represent angle down from the zenith, angle around the origin and radius from the origin respectively. In the Go Motion library, the choice of which representation to use can be made for convenience. However, within ROS tf library, Cartesian representations predominate and will be assumed unless otherwise specified.

Orientations are vectors that indicate how something is rotated. In the three-dimensional world, three numbers are necessary to indicate orientation. Go Motion supports orientation vectors in several representations: roll, pitch and yaw; Euler angles; quaternions; rotation vectors and rotation matrices. Some of these representations use more than three numbers, exploiting redundancy to make calculations with these representations more efficient. For example, a quaternion uses four numbers, and a rotation matrix uses nine numbers.

Vectors are usually written as a column of numbers enclosed in vertical bars, like this:

|  |  |  |
| --- | --- | --- |
|  | - a vector depicted in its column form | (1) |

This can be unwieldy in text documentation, so vectors may also be written as a row of numbers enclosed in braces, like this:

|  |  |  |
| --- | --- | --- |
|  | - a vector depicted in its row form | (2) |

The interpretation of a vector depends on the quantity it represents. The vector shown above could mean a translation of 1, 2 and 3 units in the x, y and z directions if the vector were a Cartesian position, or a rotation of 1, 2 and 3 units around the x, y and z directions if the vector were an orientation in roll, pitch and yaw.

Both position and orientation are needed to fully describe where something is and how it is rotated. The combination of position and orientation is called a 'pose'. Poses can be shown in row form like this pose representing a Cartesian position of (1 2 3) and an orientation in roll, pitch and yaw of (30 -30 90):

|  |  |  |
| --- | --- | --- |
|  | ( 1 2 3 ; 30 -30 90) | (3) |

A semicolon is used to separate the position from the orientation.

### Reference Frames

Regardless of the representation chosen, the numbers that indicate position and orientation of an object depend on the established origin. Several origins may be established for convenience, for example one fixed in the world or one that moves with a tool. These origins may differ from each other in both position and orientation. The establishment of the position and orientation of an origin is a 'reference frame'. The term 'coordinate frame' is used interchangeably with 'reference frame'.

When several reference frames are being used, they are denoted as identifiers in braces, for example,

|  |  |  |
| --- | --- | --- |
|  | {A} - a reference frame called 'A'.  {world} - the world reference frame.  {tool} - the world reference frame. | (4) |

To convert the representation of a pose in one reference frame to its representation in another, one needs to know the position and orientation of one origin with respect to the other. This difference between the two origins is called a 'transform'.

Poses and transforms are similar things; both include position and orientation. Whether something is a pose or a transform depends on how one is using it. Poses are used to indicate the position and orientation of things with respect to an established reference frame. Transforms are used to indicate the position and orientation of reference frames with respect to other reference frames. If a 'thing' happens to be a reference frame, its pose is its transform.

## Math Nomenclature

Assume the letter P is used to denote positions and the letter **R** is used to denote orientations. Then, trailing subscripts denote the identity of quantities, for example,

|  |  |  |
| --- | --- | --- |
|  | Phand - the position of the hand | (5) |

.

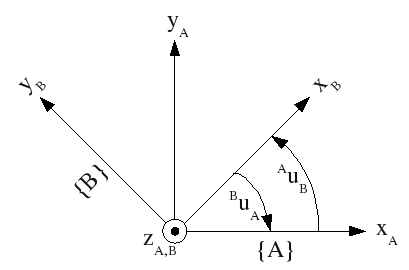
or

|  |  |  |
| --- | --- | --- |
|  | Rhead - the orientation of the head | (6) |

Leading superscripts denote the reference frame in which the quantity is expressed, for example,

|  |  |  |
| --- | --- | --- |
|  | APhand - the position of the hand with respect to the {A} reference frame  BRhead - the orientation of the head with respect to the {B} reference frame | (7) |

The figure below shows a rotation of reference frame {B} with respect to reference frame {A} by an angle **u**. Note that the rotation can be viewed as a rotation of {B} with respect to {A}, denoted AuB, or as a rotation of {A} with respect to {B}, denoted BuA. The heads of arrows are attached to the 'of the' frames, while the tails of arrows are based on the 'with respect to' frames. Angles are taken as positive according the right hand rule, so in this figure AuB is a positive number of about 45 degrees, while BuA is a negative number of the same magnitude.



*Figure* 1.

Transforms from one reference frame to another are denoted with leading subscripts and superscripts. The leading subscript denotes the original frame, and the leading superscript denotes the new frame. If a transform is purely rotation, it is denoted with an **R**, for example,

|  |  |  |
| --- | --- | --- |
| BRA | a rotation from the {A} frame to the {B} frame | (8) |

If a transform includes both a rotation and a translation, it is denoted with a T, for example,

|  |  |  |
| --- | --- | --- |
| worldTtool | a transform from the {tool} frame to the {world} frame | (9) |

To convert a quantity from one frame to another, pre-multiply the quantity by the transform. Algebraically, the leading subscript of the transform must match the leading superscript of the quantity, for example,

|  |  |  |
| --- | --- | --- |
|  | BP = BRA \* AP | (10) |

This is read, 'the position with respect to the B frame is the rotation from the A frame to the B frame times the position with respect to the A frame.' Another example is,

|  |  |  |
| --- | --- | --- |
|  | worldP = worldTtool \* toolP | (11) |

This is read, 'the position with respect to the world frame is the transform from the tool frame to the world frame times the position with respect to the tool frame.'

One can generate a rotational transform by defining an orientation vector with the appropriate angles. If using a roll, pitch and yaw representation, the rotational transforms in the figure above (each about the **z** axis) is a yaw:

|  |  |  |
| --- | --- | --- |
|  | = ARB | (12) |
|  | = BRA | (13) |

and negative angles work as well, for

|  |  |  |
| --- | --- | --- |
|  | = ARB | (14) |

## Coordinate frames

Coordinate frames are used to represent a series of transformation from the base robot frame into the last robot frame, known as wrist, end effector or tool frame. Terminology for understanding the motion control is based on Craig Chapter 1. [CRIAG]

A Class which provides coordinate transforms between any two frames in a system. This class provides a simple interface to allow recording and lookup of relationships between arbitrary frames of the system. libTF assumes that there is a tree of coordinate frame transforms which define the relationship between all coordinate frames. For example your typical robot would have a transform from global to real world. And then from base to hand, and from base to head. But Base to Hand really is composed of base to shoulder to elbow to wrist to hand. libTF is designed to take care of all the intermediate steps for you. Internal Representation libTF will store frames with the parameters necessary for generating the transform into that frame from it's parent and a reference to the parent frame. Frames are designated using an std::string 0 is a frame without a parent (the top of a tree) The positions of frames over time must be pushed in. All function calls which pass frame ids can potentially throw the exception tf::LookupException / class Transformer

For a 6 degree of robot with its base frame at (0,0,0) and the last frame at the wrist the following series of transformation define the robot.

|  |  |  |
| --- | --- | --- |
|  | 0T6 = 0T1 x 1T2 x 2T3 x 3T4 x 5T6 | (15) |

Coordinate frames are important if one needs to understand how to represent a robot with or without a tool or with different tools and/or in different positions in the world coordinate frame. Using a transformation one can move the robot around in the world coordinate system.

|  |  |  |
| --- | --- | --- |
|  | WTEE =WT0 x 0T6 x 6TEE | (16) |

Where the terminology is WT0 is the world to base frame transformation, 0T6 is the robot base to wristtransformation, and 6TEE is the wrist to tool transformation. Assuming is in the world coordinate system, the kinematic forward and inverse (FK and IK) solver we used, was ikfast.

The tool transform is specified as the position and orientation of the tool's end control point, ECP, with respect to the kinematic control point, KCP.

The kinematics functions relate the joints to the "kinematic control point" KCP. The forward kinematics calculate the KCP from the joint values, and the inverse kinematics calculate the joint values from the KCP.

Often one would like to add another transform from the KCP to an application "end control point" ECP. This is commonly called the "tool transform," with the KCP called the "wrist frame" and the ECP called the "tool frame". As tools are placed on the manipulator, the tool transform changes. 0TK is the KCP. This is the frame of the kinematics functions, where the 0 means the world coordinate system.

* 0TE is the ECP. This is the frame of motion control, program coordinates, position limits, home position and the display.
* KTE is the tool transform. The tool transform is specified as the position and orientation of the tool's end control point, ECP, with respect to the kinematic control point, KCP,

|  |  |  |
| --- | --- | --- |
|  |  | 17 |

KTE = | KRE x KPEorg|

i.e., the position and orientation of the tool tip expressed with respect to the kinematic control point (aka wrist frame).

To go from the ECP to the KCP, we postmultiply the ECP by the inverse tool transform to get the KCP:

|  |  |  |
| --- | --- | --- |
|  | 0TK = 0TE \* ETK | 18 |

that is,

|  |  |  |
| --- | --- | --- |
|  | ECP \* inverse tool transform = KCP( | 19 |

or in code as go\_pose\_pose\_mult(&ECP, &tool\_transform\_inv, &KCP). Results from the forward kinematics functions are in the KCP, and must be transformed into the ECP when sent out as status. To go from the KCP to the ECP, we postmultiply the KCP by the tool transform to get the ECP:

|  |  |  |
| --- | --- | --- |
|  | 0ET = 0KT \* KET | (20) |

that is,

|  |  |  |
| --- | --- | --- |
|  | KCP \* tool transform = ECP | (21) |

or in code as go\_pose\_pose\_mult(&KCP, &tool\_transform, &ECP). Changing the tool transform is tricky. Even the motion queue is empty and the controller is holding the ECP constant, changing the tool transform will cause a jump in the KCP and a corresponding jump in the joint values coming out of the inverse kinematics. To handle this, we need to change the ECP when changing the tool transform:

|  |  |  |
| --- | --- | --- |
|  | 0KT \* KEnewT = 0EnewT  , KCP \* new tool transform = new ECP | (22) |

When the tool is changed, you would see a jump in the displayed position, but no jump in actual position. To avoid inconsistencies between points on the motion queue in various ECPs, the tool transform can only be changed when the motion queue is empty. The configuration state table for changing the tool transform ensures this, then updates the queue position with the new ECP.

The ECP is in the traj status buffer as 'cmd\_position'. The KCP is in the traj status buffer as 'kcp'. The tool transform and inverse are in the traj settings buffer as 'tool\_transform' and 'tool\_transform\_inv'. To convert from a pose A in the ECP to the pose in the KCP, premultiply by the tool transform:

|  |  |  |
| --- | --- | --- |
|  | KA = KET \* EA | (23) |

## URDF

The Unified Robot Description Format (URDF) is an XML specification to describe a robot. URDF is a general purpose robot description specification, but is limited to only tree structures representations, ruling out all parallel robots. Also, URDF assumes the robot consists of rigid links connected by joints; flexible elements are not supported. URDFs does not support multiple groups of collision bodies. The URDF is intended to only represent the actual robot's properties, and not collisions used for external things like controller collision checking. In a URDF, the <visual> elements should be as accurate as possible to the real robot, and the <collision> elements should still be a close approximation, albeit with far fewer triangles in the meshes.

The URDF specification covers:

* Kinematic and dynamic description of the robot
* Visual representation of the robot
* Collision model of the robot

Each URDF file contains:

The robot base link

The link element describes a rigid body with an inertia, visual features,

The joint element describes the kinematics and dynamics of the joint and also specifies the safety limits of the joint.

All actuators on the robot, whether they are mounted on the robot itself or on an actuator link.

Any sensors or effectors mounted directly on an actuator link

Any other items, such as sensors or effectors mounted directly on the robot (or attached via a toolchanger), are not present in the autogenerated URDF file.

### Structure

Each robot starts with a base\_link, representing the base or main body item. Robotic arms have an invisible base\_link of zero size. Actuators, sensors, and effectors are mounted either on the base link or on an actuator link.

### Actuators

Actuators consist of a sequence of links, mounted on their parent with a joint [actuator\_name]\_mount. The links are named [actuator\_name]\_link0, [actuator\_name]\_link1... etc., while the joints are named [actuator\_name]\_joint\_1, [actuator\_name]\_joint\_2... etc. These joints are revolute joints and have limits specified in the file.

The URDF generator gets joint base poses from a USARSim GEO message, and assumes that each link spans the space from its parent joint to its child. For the last link in the arm, which has no child joint, the URDF generator uses the "TipOffset" parameter in the GEO message to determine the link's end point.

### Sensors and Effectors

Each sensor or effector is represented in the URDF file as a link with the item name. They are mounted on their parent actuator with a fixed joint named [item\_name]\_mount. Sensor coordinate frames are located at the sensor base position, but effector coordinate frames are positioned at the effector tip (with the z-axis pointing in the direction of the link the effector is mounted on). The effector tip location is determined by the "TipOffset" parameter in the GEO message.

### Using the URDF File to Publish a TF Tree

The robot\_state\_publisher node can load an URDF file to publish the TF tree for a robot in a given state. The state publisher requires that every joint in the file be published in a sensor\_msgs::JointState message to the joint\_states topic, so the main USARSim ROS node automatically publishes joint states for all actuators, as well as for any effectors mounted on actuator links.

## TRAJ Trajectory Planning Algorithms

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 occur 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 a1 at time t1 following the jerk (change in acceleration per unit time) j0. Phase 2 is an acceleration phase, with constant acceleration a1 throughout. Phase 3 is a jerk phase (or de-jerk phase) with constant (negative) jerk slowing down the acceleration from a1 to 0. Phase 4 is a constant speed phase at speed v3. Phase 5 is a constant-jerk counterpart to phase 3, where the deceleration varies smoothly from 0 to -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 -a1 to 0 and motion stops.

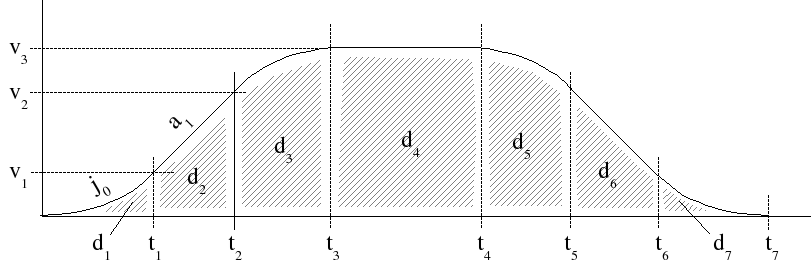
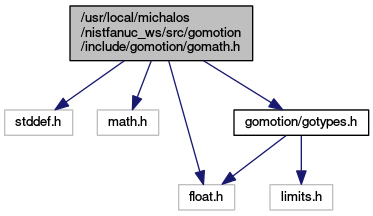


Figure 1 Constant jerk velocity profiling.

Declarations for pose math functions.

#include <stddef.h>  
#include <math.h>  
#include <float.h>  
#include "[**gomotion/gotypes.h**](file:///X:\src\github\johnmichaloski\ROS\nistfanuc_ws\src\gomotion\doc\doxygen\html\d7\d44\gotypes_8h_source.html)"

Include dependency graph for gomath.h:



**Trajectory Planning Algorithms**

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 *a1* at time *t1* following the jerk (change in acceleration per unit time) *j0*. Phase 2 is an acceleration phase, with constant acceleration *a1* throughout. Phase 3 is a jerk phase (or de-jerk phase) with constant (negative) jerk slowing down the acceleration from *a1* to 0. Phase 4 is a constant speed phase at speed *v3*. Phase 5 is a constant-jerk counterpart to phase 3, where the deceleration varies smoothly from 0 to *-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 *-a1* to 0 and motion stops.

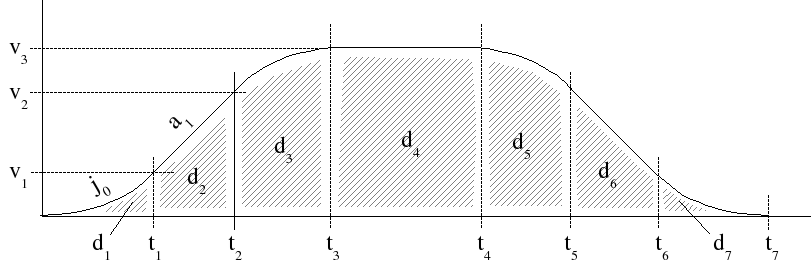
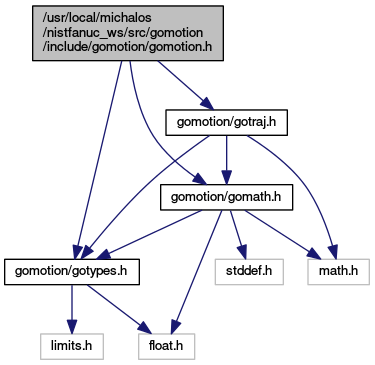


Figure 1. Constant jerk velocity profiling.

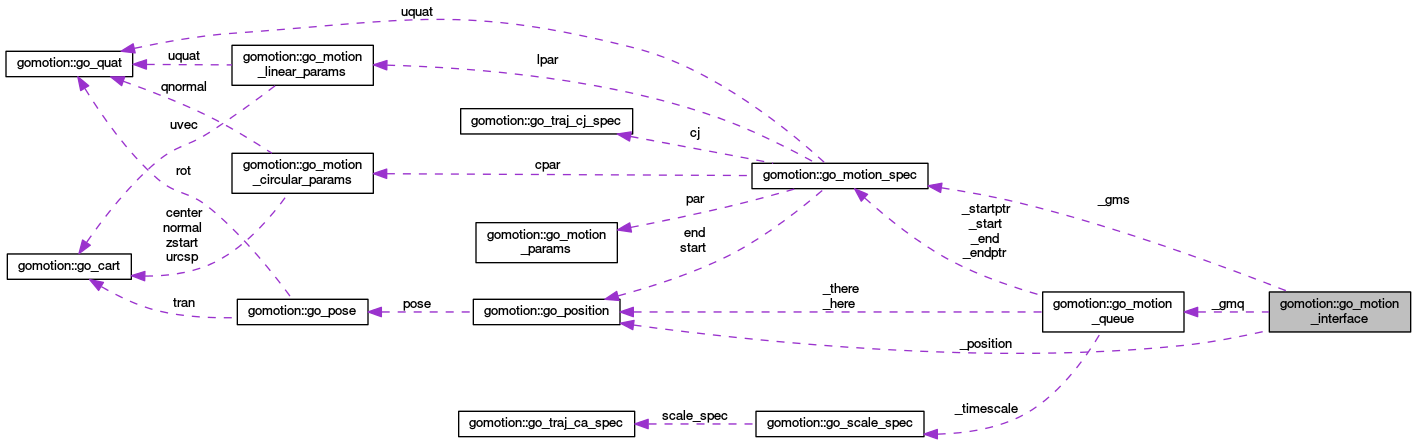
Declarations for motion queue manipulation. [More...](file:///X:\src\github\johnmichaloski\ROS\nistfanuc_ws\src\gomotion\doc\doxygen\html\d3\df0\gomotion_8h.html#details)

#include "[**gomotion/gotypes.h**](file:///X:\src\github\johnmichaloski\ROS\nistfanuc_ws\src\gomotion\doc\doxygen\html\d7\d44\gotypes_8h_source.html)"  
#include "[**gomotion/gomath.h**](file:///X:\src\github\johnmichaloski\ROS\nistfanuc_ws\src\gomotion\doc\doxygen\html\d7\d2c\gomath_8h_source.html)"  
#include "[**gomotion/gotraj.h**](file:///X:\src\github\johnmichaloski\ROS\nistfanuc_ws\src\gomotion\doc\doxygen\html\d7\dc0\gotraj_8h_source.html)"

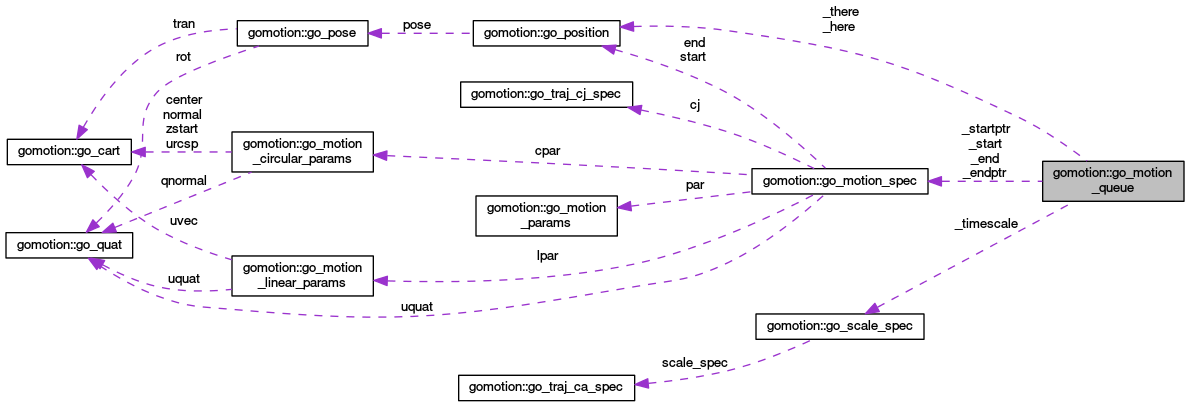
Include dependency graph for gomotion.h:



# Gomotion ROS Package

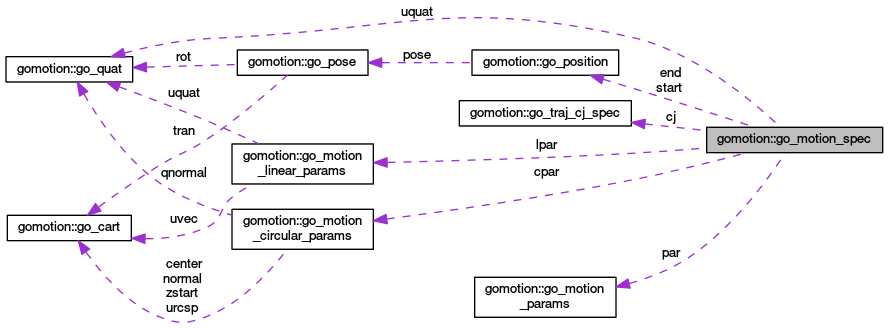


Collaboration diagram for gomotion::go\_motion\_queue:



#include <[**gomotion.h**](file:///X:\src\github\johnmichaloski\ROS\nistfanuc_ws\src\gomotion\doc\doxygen\html\d3\df0\gomotion_8h_source.html)>

Collaboration diagram for gomotion::go\_motion\_spec:



Setpoints to Servo are always in Servo's original startup coordinate system (CS). As an example, assume Servo starts out at 5.5. Servo's \a input\_latch is some point in the original CS, say 17. That is, the joint moved from 5.5 to 17 and there it saw a home condition. After the stop, the joint will have moved some extra distance, say to 17.1. Traj has a nominal value for the joint home, say 29. Traj wants to call 29 what Servo calls 17.

The difference between Servo's home in the original CS, here 17, and Traj's nominal home, here 29, is an offset, here 12. Traj needs to add 12 to Servo's position values to get values in Traj's homed CS.

When a joint is homed, Traj needs to add the offset to Servo positions when reading them, and subtract this offset to setpoints to Servo when writing them.

Servo will guarantee that \a inputOffset and \a homed are set simultaneously, so it's safe for Traj to check the \a homed flag and then add/subtract inputs/outputs based on that.

The homing sequence is as follows: Traj should begin by sending \a servo\_cmd\_servo commands with a constant setpoint to hold position, and the \a home flag cleared. Traj waits until the \a homing flag is cleared in the joint servo status. The \a homing flag is simply an echo of Traj's \home flag. Any homing action in Servo will be abandoned. When Servo sees the \a home flag set, and its \a homing flag clear, it requests a homing action if necessary from the external interface, and clears its \a homed flag.

When Servo sees the homing condition met by the external interface, it saves the latched position to the \a input\_latch in its status, sets the \a homed flag and leaves the \a homing flag set so that it won't re-initiate a home. When Traj sees the \a homed flag set for a joint, it stops the motion for that joint.

Gomotion world trajectory generation moves the mechanism to the end goal pose, which is expressed in world coordinates. If a time in seconds is given and is positive, the move will be scaled to take that amount of time if possible or longer if the system is constrained by max velocity, acceleration and jerk. If time is not positive, then the motion parameters tv, ta, tj, rv, ra and rj are used for the translational and rotational vel, accel and jerk, respectively.

The following d, v, a, j are the distance, velocity, acceleration and jerk for each joint, in meters for prismatic joints, and radians for rotational joints. The distance value 'd' is absolute, and can be positive or negative. The parameter values 'v', 'a' and 'j' must be positive

The ROS package gomotion is a library that you link against when you are compiling. You start the use of ROS package gomotion by including it in your Package.xml declaration file, for example, the following XML snippet is necessary to include the gomotion package:

?xml version="1.0" encoding="UTF-8"?>

<package>

<name>yourpackage</name>

<build\_depend>gomotion</build\_depend>

. . .

<run\_depend>gomotion</run\_depend>

You will need to modify the CMakeLists.txt that the ROS build manager "catkin" uses to build your ROS package. Below you will need to find the gomotion package and let the gomotion library be found using CATKIN\_DEPENDS, and not explicitly linking against the gomotion library. Below is a snippet of the addition to CMakeLists.txt to make gomotion library part of the include and link in the make of your package. Note, I had to add ${catkin\_LIBRARIES} AND remove the reference to the shared library libgomotion.so.

cmake\_minimum\_required(VERSION 2.8.3)

project(yourpackage)

find\_package(catkin REQUIRED COMPONENTS

. . .

gomotion

)

catkin\_package(

INCLUDE\_DIRS

include

# include/${PROJECT\_NAME}/CrclXsd

LIBRARIES

CATKIN\_DEPENDS

. . .

Gomotion

)

target\_link\_libraries(yourpackage

# I had to add ${catkin\_LIBRARIES} AND remove the reference to the shared library

# libgomotion.so

${catkin\_LIBRARIES}

${Boost\_LIBRARIES}

)

Once you have the gomotion dependencies in place, you are ready to use the trajectory generator. First, you include the gomotion header file and then declare a GoMotion trajectory generator.

#include <gomotion/gomove.h>

GoMotion go;

Gomotion can handle joint or Cartesian pose. In ROS, there are several ways to describe a kinematic pose, but we will use tf::Pose from the transform package.

tf::Pose finalpose = Conversion::Convert<geometry\_msgs::Pose, tf::Pose>(cmd.finalpose);

tf::Pose goalpose = finalpose \* \_nc->invGripperPose();

If 'totalt' is positive, then it is the time for the move. Otherwise, it's automatically computed from the vel, acc and jerk, the usual case. Here we set 'totalt' to zero to get the usual case.

At the heart of the gomotion trajectory generation is a motion queue that maintains a sequential list poses or joint values to move through. These motions are represented by the internal gomotion

The queue motion can handle either joints or pose, so it depends upon whether you are doing joint interpolation or world coordinate interpolation. In the ROS gomotion package, specifying the gomotion coordinate system is done with either a call to InitJoints or a call to InitPose, which get mapped internally into gomotion queue set\_type()). Once you have committed to Depending on the ROS gomotion init call, the internal gomotion queued is filled with either joints[] or poses accordingly.

gomotion specification

For joint specified motion, either the motion specs (phase times, distances) for each joint, or total time allotted to all the joints motion is and in so doing, the maximum time computed and retained. This maximum time will be used to scale all the joint motions so that they arrive at the same time (if specified), and internally in the gomotion will be the overall time 'totalt' for the motion.

struct GoMotionParams {

GoMotionParams(double \_vel, double \_acc, double \_jerk) :

vel(\_vel), acc(\_acc),jerk(\_jerk)

{

}

double vel; /\*< max vel for each motion \*/

double acc; /\*< max accel for each motion \*/

double jerk; /\*< max jerk for each motion \*/

};

class GoMotion {

public:

GoMotion();

int Init(JointState here, double deltat);

int InitPose(tf::Pose here, tf::Pose there,

double deltat,

gomotion::GoMotionParams tparams,

gomotion::GoMotionParams rparams);

int InitJoints(JointState here, JointState there,

double deltat,

gomotion::GoMotionParams params);

int InitUJoints(JointState here, JointState there,

double deltat,

gomotion::GoMotionParams params);

void AppendPose(tf::Pose);

tf::Pose NextPose();

JointState NextJoints();

bool IsDone();

void InitStop(); // Then use next pose or net joints

protected:

boost::shared\_ptr<go\_motion\_interface> pgm;

size\_t num\_joints;

};

int GoInterpreter::ParseJointCommand(RCS::CanonCmd cmd, RCS::CanonCmd &outcmd,

RCS::CanonWorldModel instatus, RCS::CanonWorldModel &outstatus) {

try {

cmd.joints = \_kinematics->UpdateJointState(cmd.jointnum, instatus.currentjoints, cmd.joints);

if (cmd.CommandNum() != \_lastcmdid) {

\_lastcmdid = cmd.CommandNum();

\_go->InitJoints(instatus.currentjoints, cmd.joints, this->\_nc->CycleTime(),

gomotion::GoMotionParams(1.0, 10.0, 100.0)); // 1 meter/sec

}

outcmd = cmd;

outcmd.joints = \_go->NextJoints();

} catch (std::exception & e) {

LOG\_DEBUG << "Exception in GoInterpreter::ParseCommand() thread: " << e.what() << "\n";

cmd.crclcommand = CanonCmdType::CANON\_STOP\_MOTION;

cmd.opmessage = e.what();

cmd.stoptype = CanonStopMotionType::NORMAL;

outcmd = cmd;

return CanonStatusType::CANON\_ERROR;

}

if (\_go->IsDone())

return CanonStatusType::CANON\_DONE;

else

return CanonStatusType::CANON\_WORKING;

}

int GoInterpreter::ParseWorldCommand(RCS::CanonCmd cmd, RCS::CanonCmd &outcmd,

RCS::CanonWorldModel instatus, RCS::CanonWorldModel &outstatus) {

try {

tf::Pose finalpose = Conversion::Convert<geometry\_msgs::Pose, tf::Pose>(cmd.finalpose);

// Need to subtract off tool offset from robot wrist or final tcp

//tf::Pose goalpose = finalpose \* \_nc->invGripperPose();

// Need to translate goal pose from world coordinates into robot coordinates

//goalpose = \_nc->invBasePose() \* goalpose;

//tf::Pose goalpose = finalpose;

tf::Pose lastpose=\_kinematics->FK(\_nc->status.currentjoints.position);

tf::Pose curpose=\_nc->basePose() \*

lastpose \*

\_nc->gripperPose();

if (cmd.CommandNum() != \_lastcmdid) {

\_lastcmdid = cmd.CommandNum();

\_go->InitPose(curpose, finalpose, this->\_nc->CycleTime(),

gomotion::GoMotionParams(1.0, 10.0, 100.0),

gomotion::GoMotionParams(.1, 1.0, 10.0)); // 1 meter/sec

// \_go->AppendPose(lastpose); // NO WAY~!

}

tf::Pose goalpose =\_nc->invBasePose() \* finalpose \* \_nc->invGripperPose();

tf::Pose gopose = \_go->NextPose() ;

tf::Pose nextpose = \_nc->invBasePose() \* gopose \* \_nc->invGripperPose();

JointState goaljoints;

goaljoints.position = \_kinematics->IK(goalpose, Subset(\_nc->status.currentjoints.position, \_nc->Kinematics()->NumJoints()));

cmd.joints.position = \_kinematics->IK(nextpose, Subset(\_nc->status.currentjoints.position, \_nc->Kinematics()->NumJoints()));

\_kinematics->IK(lastpose, Subset(\_nc->status.currentjoints.position, \_nc->Kinematics()->NumJoints()));

WORLDLOG << " Goal Joints =" << VectorDump<double>(goaljoints.position).c\_str() << "\n";

WORLDLOG << " Current Joints =" << VectorDump<double>(\_nc->status.currentjoints.position).c\_str() << "\n";

WORLDLOG << " Commanded Joints=" << VectorDump<double>(cmd.joints.position).c\_str() << "\n";

cmd.joints.name = \_kinematics->JointNames();

cmd.crclcommand = CanonCmdType::CANON\_MOVE\_JOINT;

outcmd = cmd;

if (\_go->IsDone())

return CanonStatusType::CANON\_DONE;

else

return CanonStatusType::CANON\_WORKING;

} catch (std::exception & e) {

LOG\_DEBUG << "Exception in GoInterpreter::ParseCommand() thread: " << e.what() << "\n";

cmd.crclcommand = CanonCmdType::CANON\_STOP\_MOTION;

cmd.opmessage = e.what();

cmd.stoptype = CanonStopMotionType::NORMAL;

outcmd = cmd;

return CanonStatusType::CANON\_ERROR;

}

}

<http://moveit.ros.org/about/>

R. Diankov and J. Kuffner, “Openrave: A planning architecture for autonomous robotics,” Tech. Rep. CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA, July 2008

“Openrave, ik fast module, openrave documentation.” http: //openrave.org/docs/latest\_stable/openravepy/ ikfast/#ikfast-the-robot-kinematics-compiler, Cited January 2014

1. Gerkey, B., Vaughan, R., Howard, A.: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In: 11th International Conference on Advanced Robotics (ICAR 2003), Coimbra, Portugal, pp. 317–323 (2003)
2. Gazebo, <http://playerstage.sourceforge.net/index.php?src=gazebo>
3. Microsoft Robotics Studio, <http://msdn.microsoft.com/robotics>
4. The Player Project, <http://playerstage.sourceforge.net/wiki/Main/_Page>
5. USARSim, <http://sourceforge.net/projects/usarsim/>
6. Webots, <http://www.cyberbotics.com/>
7. OpenHRP, <http://www.is.aist.go.jp/humanoid/openhrp/>
8. OpenGRASP, <http://opengrasp.sourceforge.net/>
9. Miller, A., Allen, P.: GraspIt!: A versatile simulator for robotic grasping. IEEE Robotics & Automation Magazine 11, 110–122 (2004)[CrossRef](http://dx.doi.org/10.1109/MRA.2004.1371616)
10. Diankov, R., Kuffner, J.: OpenRAVE: A Planning Architecture for Autonomous Robotics. Technical report, Robotics Institute, Pittsburgh, PA (2008)
11. PAL (Physics Abstraction Layer), <http://www.adrianboeing.com/pal>
12. COLLADA, [http://collada.org](http://collada.org/)
13. Khronos Group, <http://www.khronos.org/>
14. Boeing, A., Bräunl, T.: Evaluation of real-time physics simulation systems. In: 5th international Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia (GRAPHITE 2007), Perth, Australia, pp. 281–288 (2007)
15. Blender, [http://Blender.org](http://blender.org/)
16. Gaiser, I., Schulz, S., Kargov, A., Klosek, H., Bierbaum, A., Pylatiuk, C., Oberleand, R., Werner, T., Asfour, T., Bretthauer, G., Dillmann, R.: A new anthropomorphic robotic hand. In: IEEE/RAS International Conference on Humanoid Robots (Humanoids), pp. 418–422 (2008)
17. Illusoft: Blender Collada Plugin, <http://colladablender.illusoft.com/cms/>
18. Blender - Google Summer of Code (2009), <http://www.blendernation.com/blender-google-summer-of-code-2009/>
19. OpenCOLLADA, <http://www.opencollada.org/>
20. Otto Bock, <http://www.ottobock.de/cps/rde/xchg/ob_de_de/hs.xsl/384.html>
21. GRASP Project, <http://www.csc.kth.se/grasp/>
22. OpenRAVE website, <http://openrave.programmingvision.com/>
23. Björkmann, M., Kragic, D.: Active 3D scene segmentation and Detection of Unknown Objects. In: International Conference on Robotics and Automation (ICRA 2010), Anchorage, Alaska, USA (2010)
24. Papazov, C., Burschka, D.: Stochastic Optimization for Rigid Point Set Registration. In: Bebis, G., Boyle, R., Parvin, B., Koracin, D., Kuno, Y., Wang, J., Pajarola, R., Lindstrom, P., Hinkenjann, A., Encarnação, M.L., Silva, C.T., Coming, D. (eds.) ISVC 2009. LNCS, vol. 5876, pp. 1043–1054. Springer, Heidelberg (2009)[CrossRef](http://dx.doi.org/10.1007/978-3-642-10331-5_97)
25. Richtsfeld, M., Vincze, M.: Grasping of Unknown Objects from a Table Top. In: ECCV Workshop on ’Vision in Action: Efficient strategies for cognitive agents in complex environments’, Marseille, France (2008)
26. Przybylski, M., Asfour, T., Dillmann, R.: Unions of Balls for Shape Approximation in Robot Grasping. To appear in IROS (2010)
27. Asfour, T., Regenstein, K., Azad, P., Schröder, J., Vahrenkamp, N., Dillmann, R.: ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In: IEEE/RAS International Conference on Humanoid Robots (Humanoids), pp. 169–175 (2006)
28. Asfour, T., Welke, K., Azad, P., Ude, A., Dillmann, R.: The Karlsruhe Humanoid Head. In: IEEE/RAS International Conference on Humanoid Robots (Humanoids), pp. 447–453 (2008)
29. Kasper, A., Becher, R., Steinhaus, P., Dillmann, R.: Developing and Analyzing Intuitive Modes for Interactive Object Modeling. In: International Conference on Multimodal Interfaces (2007), <http://i61www.ira.uka.de/ObjectModels>
30. ROS, <http://www.ros.org/wiki/>

To stop, call go\_traj\_cj\_stop for each joint, or tran/rot motion, so that each stops as fast as it can. The longest to stop will set the revised stop time, and each will be extended to stop at that longest time. Note that the specptr values are incremental for the move, as the original distance passed to go\_traj\_cj\_compute was the incremental distance.