Visual Servo Control of a dual-armed Baxter Robot

SHORT TECHNICAL REPORT

Stein, Matthew | July 25, 2014

Introduction

This report documents a short-term project to implement dual-arm visual servo control of the Baxter Robot conducted at Institut de Recherche en Communications et Cybernétique de Nantes at Ecole Centrale. The purpose of this report is to document the effort with sufficient technical detail to allow replication by subsequent researchers. This report will provide an overall description of the project highlighting several subtle technical challenges were encountered during this effort. A complete code listing is included in Appendix A. Parameters used by the control nodes are presented in the "launch file" shown in Appendix B.

Background

This project employed three base technologies developed by others. None of these were modified in any way by the author, rather the author used available information to configure the resources by setting parameters and selecting options. This section will briefly present these technologies giving due credit to others for their development:

Figure 1 Baxter Research Robot equipped with parallel fingered grippers

BAXTER ROBOT

This project employed an unmodified Baxter Research Robot®, (Copyright © 2008-2013 Rethink Robotics, Inc. Boston, MA 02210, http://www.rethinkrobotics.com/) in its original condition. Each of the two arms were fitted with standard parallel-fingered grippers, as shown in Figure 1. Also shown in the figure are two options for finger attachments on the parallel gripper. The right arm from the perspective of the robot is equipped with long fingers for grasping the bottle shown in the figure, the left arm shorter fingers for grasping the cup.

ROS

The standard configuration of the Baxter robot includes ROS (Robot Operating System) and open-source and widely distributed software system for controlling a wide variety of robots (Open Source Robotics Foundation, http://osrfoundation.org/). The authors installed ROS on the laptop shown in the foreground of Figure 1. ROS provides both a development environment and a real-time control environment allowing researchers to develop custom programs in a programming language (C++, python, etc.), compile and execute these programs on the robot and diagnose the resulting behavior using a set of reporting tools.

AR_POSE

"ar_pose" is part of ARToolKit, an open source image recognition package compatible with ROS and the Baxter Robot (supplied under GPL by ARToolworks, Seattle, WA 98115, http://www.artoolworks.com). Once installed and properly configured, the ar_pose package will find a specially-made tag if this tag is within the camera view and return the relative position of the tag with respect to the camera. Examples of the special tag are visible in Figure 1, and this figure also suggests their application. By attaching these tags rigidly to an object, as shown on the bottle and the cup, the robot may then employ cameras built into each of its hands to find the object. Although this report will describe the configuration and use of ar_pose in determining the position of the objects, the author takes no credit for the development of ar_pose.

Approach

Building on some preliminary efforts from master's students Alessio Capitanelli, and Andrea Nistica, the approach is to attach a small paper tag to the desired object and place the object in view of the embedded hand camera. When the ar_pose software is able to recognize the tag, it reports a relative displacement between the robot hand and the tag. This displacement is used as

an error signal driving closed loop control of the robot arm towards a goal state. Once the goal state is reached, the arm control is discontinued in favor of grasping control.

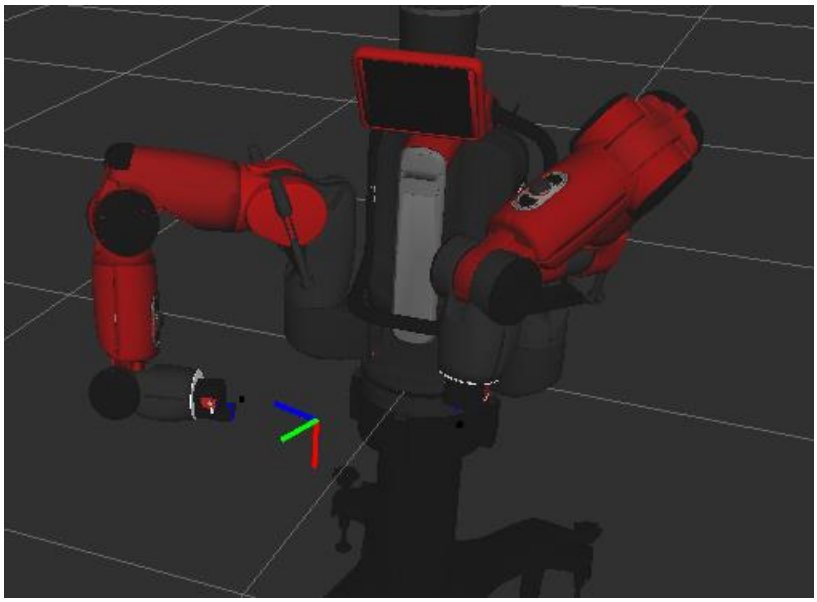


Figure 2 Rviz image of the object reference frame relative to the robot arm

Figure 2 presents a pictorial description of the control strategy. The figure is generated by Rviz, a visualization component of the ROS system that shows a live image of the robot and the robot's associated reference frame. The figure shows the robot's right arm absent the parallel fingered gripper. Note that *Figure 2* demonstrates that the robot has no internal representation of either the parallel fingered gripper or of the bottle. The red-green-blue reference frame in *Figure 2* is the estimation of location of the marker with respect to the robot returned by ar_pose. *Figure 3* shows the live image from the left hand camera corresponding to the robot position shown in *Figure 2* with the reference frame conveniently superimposed on the marker. It is evident from *Figure 3* that at least in this case, the ar_pose program has effectively identified and located the marker.

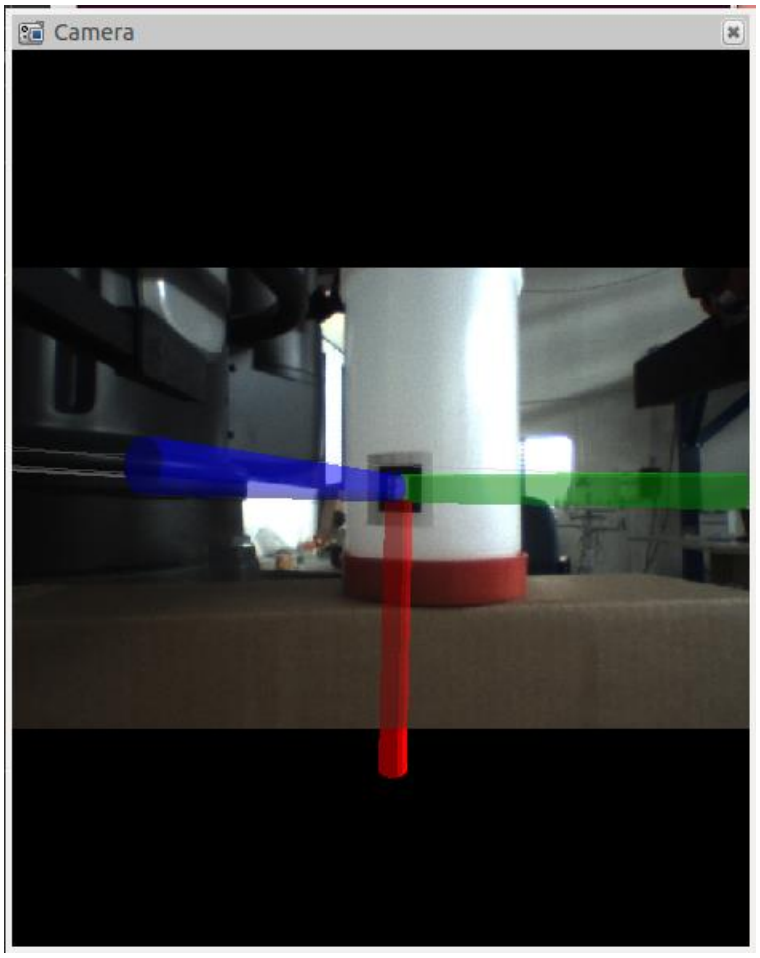


Figure 3 Live view from the left hand camera of the marker mounted on a bottle with superimposed transform.

Control Strategy

A straightforward proportional control strategy is employed in Cartesian space between the end effector reference frames as indicated in **Error! Reference source not found.**. An end effector space position error is readily computed between the origin coordinates of the two frames and an end effector space three-axis rotation is readily computed between the orientations of the two frames. We employ the arm Jacobean matrix, (also readily available from the ROS operating system) to map end effector space displacements into joint-space displacements and send these joint-space commands to the robot arm. Control gains are set by an ad hoc tuning process trading off between responsiveness and stability.

Although this approach is simple to describe, as it often is, implementation issues are another story. This section describes the overall architecture of the program and then describes significant

implementation issues encountered during development and this various techniques used to overcome these.

The control architecture consists of three nodes, as shown in Figure 4, employing one-way or "waterfall" communication. The leftmost node, "ar_pose", is the prepackaged and self-contained image recognition software. Once this node is running it continuously seeks to recognize the marker in the active camera view and publishes its result as an update of the ar_marker transform. Appendix B shows the parameters used to configure ar_pose. Most relevant are the "marker_width" set to "20.0" and the "threshold" set to "70.0", both are millimeters.

The other two nodes are custom developed and use ROS topics to communicate. The middle node, "grasp_node", responds only to the presence or absence of the ar_marker, and therefore publishes but does not subscribe to ROS topics. When a fresh ar_marker is detected it applies the end-effector space control law and requests a change in velocity by publishing to a ROS topic. When this node detects that the ar_marker has become stale it requests a halt to velocity control, resulting in the limb maintaining its current position in joint position mode. When it detects that the marker is within specified tolerance of the goal position this node requests a switch to position mode and a gripper close action.

The far right node, "state_control_node", both subscribes and publishes to ROS topics. It reads parameters specifying joint locations for pre-defined start and end locations and issues rote motion commands moving the robot to the start position. It then awaits requests from the grasp_node and implements these requests by publishing to limb and gripper control topics. It implements end-effector frame velocity commands by computing the limb Jacobean at the current limb location and multiplies the Jacobean matrix by end effector frame velocity requests to produce joint velocity requests. After clipping excessive velocity requests, it issues the request to the limb by publishing on a ROS topic.

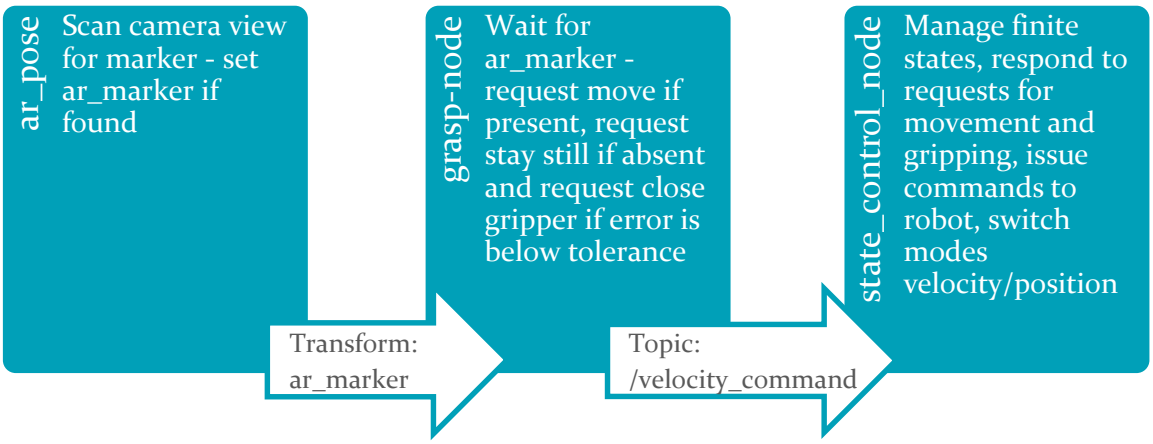


Figure 4 Pictorial description of the control nodes

Implementation Issues

FRAME PERSISTANCE

We encountered an issue with the inability to exert real-time control on the ROS transfer function manager "TF". The issue is detecting when the ar_pose program loses "sight" of the marker. Note that the quoted term "sight" is used loosely here, what it means is that the ar_pose program fails to produce a transform indicating the position of the marker. It is often the case that we can see the marker in the live camera image, but for a variety of reasons including glare and occlusion, the marker is not recognized. Ideally we would like to immediately halt the control algorithm, as continuing to move the robot based on stale information will only produce favorable results by accident. In general, it is not entirely clear what should be done in this instance. If the arm stays still there is the chance the marker will simply be detected again, or one may wish to develop some strategy to search for the marker.

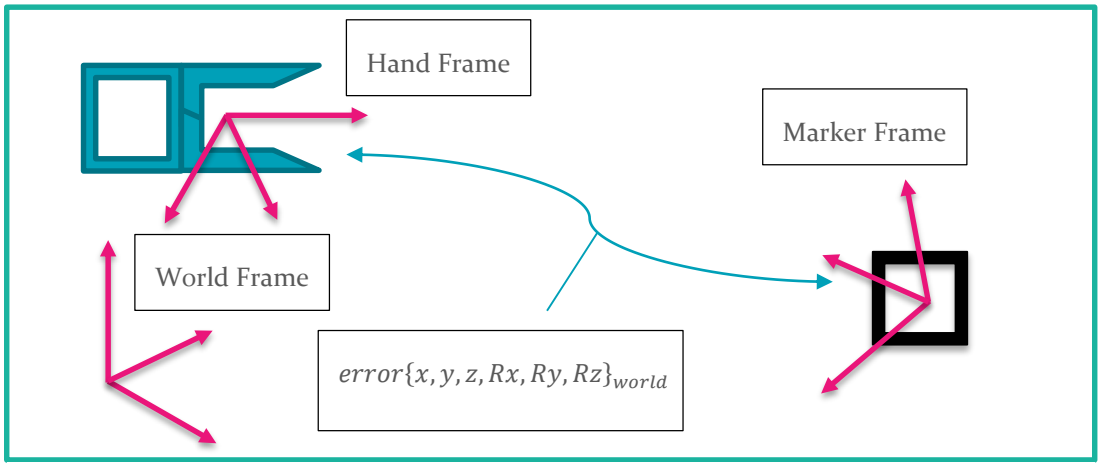


Figure 5 Pictorial description of control strategy

However, because of the distributed and asynchronous structure of ROS, the last known transform of the marker will persist in the TF manager for up to 10 seconds. Absent the fortuitous reacquisition of the marker, the robot will move in a straight line along its last path for up to 10 seconds - clearly undesirable if not outright dangerous.

One attempt to more precisely detect the absence of the marker produced surprising results. We investigated using the supplied transform time stamps, only to find that time stamps between the robot and the control computer are desynchronized. The ROS system does not have a universal time stamp that can be used to determine if the marker transform becomes stale.

As a compromise, we placed a two-second limit on the use of stale transforms. This was implemented by checking the relative age of a continuously updating robot arm transform against the most recent ar_marker frame. Should the ar_marker frame become more than two seconds old the marker is considered stale.

VELOCITY WATCHDOG

Placing the robot arm in velocity control mode introduces a well-justified safety feature that must be worked around. Should the control computer suddenly fail, a robot in velocity mode will continue to move at the last specified velocity until some limit is reached. Because this is always bad, there is a watchdog timer built into velocity control that will kill arm power if the control computer does not continuously refresh commands.

However, even when the arms are unmoving, they are still in velocity control mode with a velocity setpoint of zero. Any linux process that causes the processor to become busy may cause the arms to lose power and plunge into the table. Our work-around is to switch the robot arm to position control mode any time the robot is not actively responding to an ar_marker. Note that this does not solve the problem – the arms can still lose power in the middle of active control, this fix just decreases the observed frequency of occurrence.

TWO ARM CONTROL

We had to work though some name-space problems to allow the same nodes to operate on two arms simultaneously. The canned program ar_pose allows the specification of a parameter to set the ar_marker name, and this combined with rewriting programs to allow the setting of parameters in a launch file allowed the nodes to run simultaneously. Students Capitanelli and Nistica implemented a joint offset of $\pi/2$ depending on the arm of choice, and their implementation proved entirely appropriate. The state_control_node interacts with the topics

```
    //subscribers

nh.subscribe("/robot/joint_states",1,stateCall);
nh.subscribe("/left_command_velocity,1,velocityCall);
nh.subscribe("/robot/end_effector/left_gripper/state",1,gripperStateCall);

    //Publishers

nh.advertise<baxter_core_msgs::JointCommand>("/robot/limb/right/joint_command",1);
nh.advertise<baxter_core_msgs::JointCommand>("/robot/limb/left/joint_command",1);
nh.advertise<baxter_core_msgs::EndEffectorCommand>("/robot/end_effector/left_gripp
er/command",1);
```

Node grasp_node interacts with the following topics

```
    //Publishers

nh.advertise<grasp::xDot>("/left_command_velocity",1);
nh.advertise<grasp::xDot>("/left_error",1);
    //Transforms
comparison_frame = "/left_wrist";
pose_topic = "/left_ar_marker";
```

In all cases the string “left” and “right” are interchangeable and there was no detectable difference between the performances of the arms.

GRIPPER CONTROL

We had some initial challenges determining the necessary commands to control the parallel fingered gripper. The required sequence is shown below. Note that the CMD_RELEASE argument may be replaced with CMD_GRIP or any other available gripper command. We found the control

was indifferent to the arguments for “sequence” and “sender” but that the value of “65538” was necessary for the “id” field. Its meaning remains mysterious.

```
baxter_core_msgs::EndEffectorCommand gripperMsg;
gripperMsg.command = gripperMsg.CMD_RELEASE;
gripperMsg.id = 65538;
gripperMsg.sequence = 3;
gripperMsg.sender = "/grasp_node";

//  Gripper commands must be continuously sent until gripper has responded
gripPub.publish(gripperMsg);
```

We observed that it was only required to send the command once, however, the time required for the gripper to open is undetermined. It was thus necessary to add a listener function that awaited gripper state messages and determine gripper state from the position argument, as shown here:

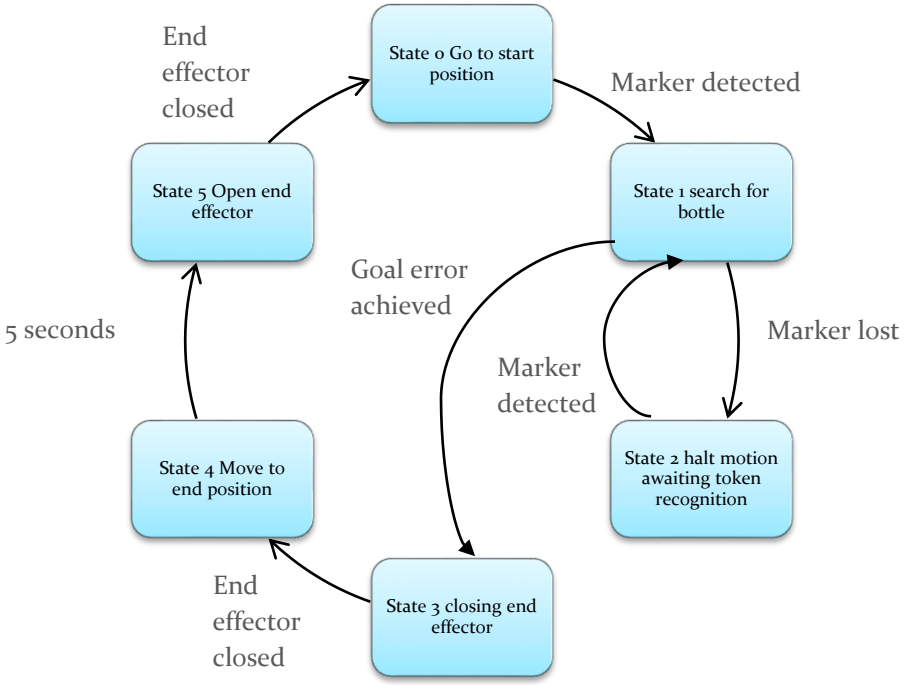
```
// This callback is run continuously as publishes of gripper State occurs at real-time rate
void gripperStateCall(baxter_core_msgs::EndEffectorState stateMsg){
    //      ROS_INFO("In gripper call, position is %f",stateMsg.position) ;
    if (stateMsg.position > 60.0) gripper_open = true;
    if (stateMsg.position < 20.0) gripper_closed = true;
}
```

Note that we have deliberately placed a gap between the positions considered open and closed and that we do not require the position argument to reach 100.0 or 0.0 to be considered open or closed.

Also note that due to a design issue with the student-designed bottle grasping attachment, symmetric installation of the fixtures required that the logic of the gripper be inverted, thus for the right arm end effector a command to open the gripper grasps the bottle and likewise a command to close the gripper releases the bottle.

DEMONSTRATION PROGRAM

We constructed a relatively simple 5-state program to demonstrate visual acquisition of the bottle.



As can be seen from the diagram, states 3 to 0 are a rote motion awaiting either the gripper to respond or five seconds to elapse. These motions behave in a completely predictable manner. All non-deterministic aspects of the demonstration program are related to acquisition of the marker and grasping of the bottle.

DEMONSTRATION PROGRAM PERFORMANCE

The success of the bottle grasping task depends heavily on the ar_pose program to detect the marker. To understand its performance, we configured the Rviz program as shown in Figure 3 and used this view to monitor the ar-pose program during operation. We observed that for the most part the program works admirably, detecting the marker and returning it's position with impressive accuracy and repeatability. We have observed, however, that the recognition sometimes fails, even when a live camera image plainly shows the marker is in view. We have noticed that the recognition process can be degraded by the absence of light, by the presence of back glare and by the presence of back clutter in the view that can potentially be mistaken for the marker. In the live camera window we see that the ar_pose program occasionally locates the marker on an extraneous object of this object was stark black/white contrast similar to the marker. We also note that back glare due to sunlight entering through inaccessible skylights deteriorates the performance, and that this deterioration can be reduced by adding directed artificial light.

The demonstration works best when ar_pose frequently returns accurate marker locations. When recognition fails the robot continues to move at the last detected velocity until the stale marker is detected as mentioned above. As a result, if marker location is intermittent than robot motion is jerky and haphazard, and this can have the effect of knocking over the bottle on approach. In

addition, the goal position of the control will place the end effector around the bottle while the marker is in plain sight of the hand camera. But we have noticed that in the presence of challenging lighting ar_pose fails to detect the marker, and in this case the arm will halt with its end effector encompassing the bottle but the grasp command will never be issued. When in this position, the demonstration program has effectively become stuck as only the reacquisition on the marker will cause the robot to move.

Conclusion

Overall the project was successful as the recorded video images show several instances of the right robot arm grasping the bottle under visual servo control. As mentioned, the demonstration works best in favorable light conditions that cause frequent and reliable detection of the marker. Under these conditions the task is successful from a surprisingly large variety of bottle starting locations and orientations and even seeks a moving bottle. In the latter case the robot exhibits a somewhat graceful and human-reminiscent motion of taking a bottle handed to it.

APPENDIX A

CODE LISTING

```
/*=========================================================//
// Author     :Alessio Capitanelli, Andrea Nisticò     //
// Modified by  : Matthew Stein                          //
// Name       :control_node                          //
// Version      :1.0.0                                //
// Date         :17/07/2014                          //
// Description  :Send velocity commands to cartesian_node//
//               calculated in order to reach desired    //
//               pose. Then switch to position mode and  //
//               grasp the object.                       //
//=========================================================*/

#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <tf/transform_broadcaster.h>
#include <eigen3/Eigen/Core>
#include <eigen3/Eigen/Dense>
#include <sensor_msgs/JointState.h>
#include <geometry_msgs/PoseStamped.h>
#include <baxter_core_msgs/JointCommand.h>
#include <grasp/xDot.h>
#include <iostream>
using namespace std ;
using namespace Eigen ;

double desX; //desired pose parameter
double desY;
double desZ;
double desXRot;
double desYRot;
```

```
double desZRot;
double desWRot;

double desRoll;
double desPitch;
double desYaw;



VectorXd desiredPoseVect(6);
VectorXd actualPoseVect(6);
VectorXd errorPoseVect(6);
//needed messages
grasp::xDot poseError;
grasp::xDot command;

//gains

double kX , kY , kZ ,kR;

//parameters
int arm;


bool graspOK;
bool valid_transform; //  Flag to shut off commands once there is no valid
transform available
bool position_reached = false ;

//param



int main(int argc, char** argv){

    //init
    ros::init(argc, argv, "control");
    ros::NodeHandle nh;
    ros::NodeHandle nh_("~");   //ROS Handler - local namespace.


    string retString;
    arm=1;  // Get arm parameter, default is right
    if( !nh_.getParam("arm",retString) ){
        ROS_INFO("Couldn't find parameter: arm") ;
        } else {
        if (retString == "left") arm=0;
        else if (retString == "right") arm=1;
        else ROS_INFO("Parameter arm did not contain left or right") ;

        ROS_INFO("Setting parameter arm to %d",arm) ;
    }

    // Setting the position and orientation tolerances for considering
    // a position as reached. Position tolerance defaults to 5 mm.
    // Orientation tolerance defaults to 5 degrees.
```

```
double position_tolerance, orientation_tolerance ;
nh_.param("position_tolerance"    ,position_tolerance   ,0.008) ;
nh_.param("orientation_tolerance",orientation_tolerance,15*M_PI/180.0) ;

string comparison_frame;
string pose_topic;
string velocity_topic;
string error_topic;


if (arm==0) {
    comparison_frame = "/left_wrist";
    pose_topic = "/left_ar_marker";
    velocity_topic = "/left_command_velocity";
    error_topic = "/left_error";

}
else {
    comparison_frame = "/right_wrist";
    pose_topic = "/right_ar_marker";
    velocity_topic = "/right_command_velocity";
    error_topic = "/right_error";
}
//parameters (desired pose in Roll Pitch Yaw)
if (arm==0) {  // Left arm parameters for small cup
    nh.param("/desiredX",desX,0.04);
    nh.param("/desiredY",desY,0.0);
    nh.param("/desiredZ",desZ,0.22);
    nh.param("/desiredXRot",desXRot,0.0);
    nh.param("/desiredYRot",desYRot,0.0);
    nh.param("/desiredZRot",desZRot,0.0);
    } else { // Right arm parameters for bottle

    nh.param("/desiredX",desX,0.05);
    nh.param("/desiredY",desY,0.0);
    nh.param("/desiredZ",desZ,0.19);
    nh.param("/desiredXRot",desXRot,0.0);
    nh.param("/desiredYRot",desYRot,0.0);
    nh.param("/desiredZRot",desZRot,0.0);
}

//gains param

//nh.setParam("/kX",1.2);
//nh.setParam("/kY",1.2);
//nh.setParam("/kZ",0.8);

//nh.setParam("/kR",0.7);

nh.setParam("/kX",1.2);
nh.setParam("/kY",1.2);
nh.setParam("/kZ",0.8);

nh.setParam("/kR",0.7);
```

```
    //state param
    nh.setParam("/grasp",false);
    nh.getParam("/grasp",graspOK);



    //publishers

    ros::Publisher commandPub = nh.advertise<grasp::xDot>(velocity_topic,1);
    ros::Publisher errorPub = nh.advertise<grasp::xDot>(error_topic,1);


    //needed objects

    tf::TransformListener listener;
    tf::StampedTransform marker;
    tf::StampedTransform time_marker;
    tf::Quaternion rotQ,rotQ90,rotQinterm;
    tf::Matrix3x3 m;

    ros::Rate rate(10);




    desiredPoseVect << desX,desY,desZ,desXRot,desYRot,desZRot;


    nh.getParam("/kX",kX);
    nh.getParam("/kY",kY);
    nh.getParam("/kZ",kZ);

    nh.getParam("/kR",kR);

    valid_transform = false;
    command.v_x = 0.0;
    command.v_y = 0.0;
    command.v_z = 0.0;

    command.w_x = 0.0;
    command.w_y = 0.0;
    command.w_z = 0.0;
    command.controlState = 0;  // This boolean commands to cartesian_node to halt
velocity control
    // and to return to position mode holding at current location


    while( ros::ok()){
        ros::spinOnce() ;

        // ar_marker is published by canned package ar_pose.  When ar_marker is
seen there
        // will be a valid transform between the wrist and the marker
        // Use this test to determine when to apply control
        if( !listener.canTransform(comparison_frame,pose_topic,ros::Time(0))){
            ROS_INFO("No valid transform from wrist to ar_marker") ;
```

```
            if (valid_transform) {

                command.v_x = 0.0;
                command.v_y = 0.0;
                command.v_z = 0.0;

                command.w_x = 0.0;
                command.w_y = 0.0;
                command.w_z = 0.0;
              //0=position freeze 1 = velocity control requested 2 = close gripper
                command.controlState = 0;  //  Message to freeze at current
location

                commandPub.publish(command);

                valid_transform = false;

            }

            //go back to locate

            }else{
            listener.lookupTransform(comparison_frame,pose_topic,ros::Time(0),
marker);

listener.lookupTransform(comparison_frame,comparison_frame,ros::Time(0),
time_marker);

            //canTransform returns true for up to 10 seconds after the most recent
ar_marker
            // causing the robot to coast at the last velocity
            // A bodge to work around this is to manually check if the ar_marker
is stqle
            if(time_marker.stamp_.sec-marker.stamp_.sec < 2 ) { // If the
ar_marker is more than 1
                // second stale stop controlling, even
                // if canTransform returns true


                rotQinterm = marker.getRotation();
                valid_transform = true;
                ROS_INFO("Fresh marker present, applying control");

                //avoid singular position
                rotQ90.setRPY(3.14,0,0);
                rotQ = rotQinterm*=rotQ90;

                m.setRotation(rotQ);


                //Control laws

                double rx,ry,rz;

                m.getRPY(rx,ry,rz); // actual rotation in R P Y
```

```
                actualPoseVect << marker.getOrigin().getX() ,
marker.getOrigin().getY() , marker.getOrigin().getZ(),rx,ry,rz;
                errorPoseVect = actualPoseVect - desiredPoseVect;

                poseError.v_x = errorPoseVect(0);
                poseError.v_y = errorPoseVect(1);
                poseError.v_z = errorPoseVect(2);
                poseError.w_x = errorPoseVect(3);
                poseError.w_y = errorPoseVect(4);
                poseError.w_z = errorPoseVect(5);
                poseError.controlState = 1;

                position_reached =
                abs(poseError.v_x) < position_tolerance      &&
                abs(poseError.v_y) < position_tolerance      &&
                abs(poseError.v_z) < position_tolerance      &&
                abs(poseError.w_x) < orientation_tolerance   &&
                abs(poseError.w_y) < orientation_tolerance   &&
                abs(poseError.w_z) < orientation_tolerance   ;

                if( !position_reached ){
                    command.v_x = kX * poseError.v_x;
                    command.v_y = kY * poseError.v_y;
                    command.v_z = kZ * poseError.v_z;

                    command.w_x = kR * poseError.w_x;
                    command.w_y = kR * poseError.w_y;
                    command.w_z = kR *1.8* poseError.w_z;
                    command.controlState = 1;
                    }else{  // position reached, call for grasp
                    command.v_x = 0.0 ;
                    command.v_y = 0.0 ;
                    command.v_z = 0.0 ;

                    command.w_x = 0.0 ;
                    command.w_y = 0.0 ;
                    command.w_z = 0.0 ;
                    command.controlState = 2 ;
                }

                commandPub.publish(command);
                errorPub.publish(poseError);

            } else {  // ar_marker has become stale, call for freeze

                command.v_x = 0.0;
                command.v_y = 0.0;
                command.v_z = 0.0;

                command.w_x = 0.0;
                command.w_y = 0.0;
                command.w_z = 0.0;
                command.controlState = 0;

                commandPub.publish(command);
```

```
            }

        }

        rate.sleep() ;

    }

    return 0;

}


/*=====================================================//
// Author   :Alessio Capitanelli, Andrea Nisticò         //
// Modified by  : Matthew Stein                         //
// Name          :approach_node                      //
// Version      :1.0.0                       //
// Date         :17/07/2014                     //
// Description  :Use the jacobian of the BAXTER robot   //
//               to set the joint velocities needed to  //
//     achieve a given elemental displacement      //
//=====================================================*/

//CPP

#include <vector>
#include <csignal>

//ROS

#include <ros/ros.h>
#include "std_msgs/Int32.h"
#include "sensor_msgs/JointState.h"
#include <baxter_core_msgs/JointCommand.h>
#include <grasp/xDot.h>
#include "baxterJac.h"
#include <baxter_core_msgs/EndEffectorCommand.h>
#include <baxter_core_msgs/EndEffectorState.h>

//INITIALIZATIONS

int arm;

int arm_i;
int controlState=0;  //0=position freeze 1 = velocity control requested 2 = close
gripper
bool joint_state_available = false;
bool gripper_open = false;
bool gripper_closed = false;
sensor_msgs::JointState lastStateMsg;
int demo_state;
```

```
long start_time;
double start_pos[7];
double end_pos[7];
// demo state: 0=start, 1 = seeking, 2=stay still, 3=end, 4=opening gripper
5=closing gripper


sig_atomic_t volatile g_request_shutdown = 0; //signal used to kill the node
//in a safe way. Avoid "dead arms" caused by a pause in sending velocity commands
when robot is
// in velocity mode
VectorXd x_dot(6), q_dot(7), q(7), curPos(7);
baxter_core_msgs::JointCommand comMsg;// Position command to hold when in position
mode

ros::Publisher pubLeft ;
ros::Publisher pubRight ;
ros::Publisher gripPub;
void mySignalHandler(int sig){
    g_request_shutdown = 1;
}
//  Initialization function for each demo state
void state0_init() {  // Velocity control at current location
       start_time = lastStateMsg.header.stamp.sec;
       comMsg.mode = comMsg.POSITION_MODE;
       for(int i=0;i<7;i++){

              comMsg.names.push_back(lastStateMsg.name[i + arm_i] );
              comMsg.command.push_back( start_pos[i]);
       }

}

void state1_init() {  // Velocity control at current location

}

void state2_init() {  // Stay still at current location
         comMsg.mode = comMsg.POSITION_MODE ;

        // set desired position to current position
        for(int i = arm_i; i < arm_i+7 ; i++){ //arm_i is set to 2 or 9 depending
on which arm

            comMsg.command.push_back(lastStateMsg.position[i]) ;
            comMsg.names.push_back(lastStateMsg.name[i]) ;

        }


}
void state3_init() {  // Velocity control at current location
       start_time = lastStateMsg.header.stamp.sec;
       comMsg.mode = comMsg.POSITION_MODE;
       for(int i=0;i<7;i++){
```

```
            comMsg.names.push_back(lastStateMsg.name[i + arm_i] );
            comMsg.command.push_back( end_pos[i]);
        }

}
void state4_init() {  // Opens the gripper

         comMsg.mode = comMsg.POSITION_MODE ;

        // set desired position to current position
        for(int i = arm_i; i < arm_i+7 ; i++){ //arm_i is set to 2 or 9 depending
on which arm

            comMsg.command.push_back(lastStateMsg.position[i]) ;
            comMsg.names.push_back(lastStateMsg.name[i]) ;

        }

    if(arm==0) {
        baxter_core_msgs::EndEffectorCommand gripperMsg;
        gripperMsg.command = gripperMsg.CMD_RELEASE;
        gripperMsg.id = 65538;
        gripperMsg.sequence = 3;
        gripperMsg.sender = "/grasp_node";

        //  Gripper commands must be continuously sent until gripper has responded
        gripPub.publish(gripperMsg);
        } else {
        baxter_core_msgs::EndEffectorCommand gripperMsg;
        gripperMsg.command = gripperMsg.CMD_GRIP;
        gripperMsg.id = 65538;
        gripperMsg.sequence = 3;
        gripperMsg.sender = "/grasp_node";

        //  Gripper commands must be continuously sent until gripper has responded
        gripPub.publish(gripperMsg);
    }
    gripper_open = false;
    gripper_closed = false;
}

void state5_init() {  // Closes
    if(arm==0) {
        baxter_core_msgs::EndEffectorCommand gripperMsg;
        gripperMsg.command = gripperMsg.CMD_GRIP;
        gripperMsg.id = 65538;
        gripperMsg.sequence = 3;
        gripperMsg.sender = "/grasp_node";

        //  Gripper commands must be continuously sent until gripper has responded
        gripPub.publish(gripperMsg);
        } else {
        baxter_core_msgs::EndEffectorCommand gripperMsg;
        gripperMsg.command = gripperMsg.CMD_RELEASE;
        gripperMsg.id = 65538;
        gripperMsg.sequence = 3;
```

```
        gripperMsg.sender = "/grasp_node";

        //  Gripper commands must be continuously sent until gripper has responded
        gripPub.publish(gripperMsg);
    }

    gripper_open = false;
    gripper_closed = false;
}

//CALLBACKS
// This topic is published by control_node
void velocityCall(grasp::xDot velocityMsg){ //uses a custom message

    x_dot(0) = velocityMsg.v_x;
    x_dot(1) = velocityMsg.v_y;
    x_dot(2) = velocityMsg.v_z;
    x_dot(3) = velocityMsg.w_x;
    x_dot(4) = velocityMsg.w_y;
    x_dot(5) = velocityMsg.w_z;
    controlState = velocityMsg.controlState;
    if(demo_state == 1 || demo_state == 2) { // ignore marker in other states
           if(controlState == 1) {
              state1_init();
              demo_state=1;

           } else if(controlState == 2){
               ROS_INFO("Received request to grip") ;
                     state5_init();
                     demo_state =5;

           } else if(controlState == 0){
                     state2_init();
                     demo_state =2;
           }
    }
}

// This callback is run continuously as published of gripper State occur at real-
time rate
void gripperStateCall(baxter_core_msgs::EndEffectorState stateMsg){
    //      ROS_INFO("In gripper call, position is %f",stateMsg.position) ;
    if (stateMsg.position > 60.0) gripper_open = true;
    if (stateMsg.position < 20.0) gripper_closed = true;
}


// This callback is run continuously as published of JointState occur at real-time
rate
void stateCall(sensor_msgs::JointState stateMsg){

    sensor_msgs::JointState state;
    lastStateMsg = stateMsg;
    joint_state_available = true;
    //setting joints in the right order to feed the jacobian
    //JointState order -> e0, e1, s0, s1, w0, w1, w2
```

```
//desired order -> s0, s1, e0, e1, w0, w1, w2

for(int i = 0; i < 2 ; i++){  // s0 and s1

    q(i) = stateMsg.position[i + arm_i + 2];
    curPos(i) = stateMsg.position[i + arm_i + 2];
    state.position.push_back(stateMsg.position[i + arm_i + 2]);
    state.name.push_back(stateMsg.name[i + arm_i + 2]);

}

for(int i = 2; i < 4 ; i++){ // e0 and e1
    q(i) = stateMsg.position[i + arm_i - 2];
    curPos(i) = stateMsg.position[i + arm_i - 2];
    state.name.push_back(stateMsg.name[i + arm_i - 2]);

}

for(int i = 4; i < 7 ; i++){ // w0, w1, w2
    q(i) = stateMsg.position[i + arm_i];
    curPos(i) = stateMsg.position[i + arm_i];
    state.name.push_back(stateMsg.name[i + arm_i]);
}

//choose arm offset

if(arm == 1){
    q(0) = q(0) - 0.79;
    q(1) = q(1) + 1.5708;
}
else if(arm == 0){
    q(0) = q(0) + 0.79;
    q(1) = q(1) + 1.5708;
}

switch (demo_state ) {
case 0:  // Move to Start
case 2:  // Stay still while seeking
case 3:  // Move to end
case 4:  // Open Gripper
case 5:  // Close Gripperr


    if(arm){
        pubRight.publish(comMsg);

    }else {

        pubLeft.publish(comMsg);
    }

 break;
 case 1: // Seeking target in velocity mode


    //compute joint velocities
```

```
        q_dot = PseudoInverse(baxterJac477(q))*x_dot;


        //normalization to avoid excessive joint velocities
        //and actuators saturation

        if ( q_dot.maxCoeff() > 0.25 ){

            q_dot = 0.20*q_dot/q_dot.maxCoeff();
        }

        //publishing
        comMsg.mode = comMsg.VELOCITY_MODE;
        for(int i = 0; i < 7 ; i++){

            double comm = q_dot(i);
            comMsg.command.push_back(comm) ;
            comMsg.names.push_back(state.name[i]) ;
        }


        if(arm){
            pubRight.publish(comMsg);
        }
        else {
            pubLeft.publish(comMsg);
        }
    break;
    }// End switch

}


int main(int argc, char** argv){


    //initialization

    ros::init(argc, argv, "cartesianMode");
    ros::NodeHandle nh;
    ros::NodeHandle nh_("~");//ROS Handler - local namespace.

    //parameters
    string retString;



    arm=1;  // default
    if( !nh_.getParam("arm",retString) ){
        ROS_INFO("Couldn't find parameter: arm") ;

        } else {
        if (retString == "left") arm=0;
        else if (retString == "right") arm=1;
        else ROS_INFO("Parameter arm did not contain left or right") ;
```

```
        ROS_INFO("Setting parameter arm to %d",arm) ;
    }

    //nh.param("/arm",arm,1);  //this parameter allows to choose the arm to use
    //(1 -> use right, default, 0 -> use left)
    string gripper_string;
    string gripper_state_string;
    string velocity_topic;
    if (arm==0) {
        velocity_topic = "/left_command_velocity";
        gripper_string = "/robot/end_effector/left_gripper/command";
        gripper_state_string = "/robot/end_effector/left_gripper/state";
    }
    else {
        velocity_topic = "/right_command_velocity";
        gripper_string = "/robot/end_effector/right_gripper/command";
        gripper_state_string = "/robot/end_effector/right_gripper/state";
    }
    //arm selection

    if (arm == 1){

        arm_i = 9;


        }else if(arm == 0){

        arm_i = 2;


        }else{

        ROS_INFO("Invalid arm parameter: only 1 (=right) or 0 (=left) allowed.");

        return 1;
    }

    // Get starting positions
    int numParams;
    for(numParams=0;numParams<7;numParams++){
        char nameString[20];
        sprintf(nameString,"joint_start_%d",numParams);
        ROS_INFO("%s",nameString);
        if( !nh_.getParam(nameString,retString) ){
            ROS_INFO("Couldn't find parameter: joint_start_%d\n",numParams) ;
            //return 1 ;
            }else{
            if ( ! (istringstream(retString) >> start_pos[numParams]) )
start_pos[numParams] = 0;
        }
    }
    // Get ending positions
    for(numParams=0;numParams<7;numParams++){
        char nameString[20];
        sprintf(nameString,"joint_end_%d",numParams);
```

```
        ROS_INFO("%s",nameString);
        if( !nh_.getParam(nameString,retString) ){
            ROS_INFO("Couldn't find parameter: joint_end_%d\n",numParams) ;
            //return 1 ;
            }else{
            if ( ! (istringstream(retString) >> end_pos[numParams]) )
end_pos[numParams] = 0;
        }
    }

    //subscribers

    ros::Subscriber subJoint = nh.subscribe("/robot/joint_states",1,stateCall);
    ros::Subscriber subComm = nh.subscribe(velocity_topic,1,velocityCall);
    ros::Subscriber gripSub =
nh.subscribe(gripper_state_string,1,gripperStateCall);

    //Publishers

    pubRight =
nh.advertise<baxter_core_msgs::JointCommand>("/robot/limb/right/joint_command",1);
    pubLeft =
nh.advertise<baxter_core_msgs::JointCommand>("/robot/limb/left/joint_command",1);
    gripPub =
nh.advertise<baxter_core_msgs::EndEffectorCommand>(gripper_string,1);

    //Signals

    signal(SIGINT,mySignalHandler);
    signal(SIGTERM, mySignalHandler);


    x_dot(0) = 0;
    x_dot(1) = 0;
    x_dot(2) = 0;
    x_dot(3) = 0;
    x_dot(4) = 0;
    x_dot(5) = 0;

    // Move the robot to a pre-defined, neutural position before falling into
control loop
    // Important here to wait while spinning until a joint state message becomes
available
    // Once this has been received we can copy the joint name strings and enable
pos_control

    ROS_INFO("Spinning while waiting for joint state");
    // Get the joint state from callback once to map joint names to numbers
    while(!joint_state_available)
    ros::spinOnce();  // wait for joint state to be available


    controlState = 0;

    ros::Rate rate(100);
```

```
    // start demo in state 0
    state0_init();
    demo_state=0;


    int times = 0;

    while(ros::ok() && !g_request_shutdown){
        ros::spinOnce();


        // demo state: 0=start, 1 = seeking, 2=stay still, 3=end, 4=opening
gripper 5=closing gripper
        if (demo_state == 0) {
            long cur_time = lastStateMsg.header.stamp.sec;

            if(cur_time - start_time > 5) {
               state4_init();
              demo_state = 4;

            }
        }
       if (demo_state == 3) {
            long cur_time = lastStateMsg.header.stamp.sec;

            if(cur_time - start_time > 5) {
               state0_init();
              demo_state = 0;

            }
        }

        if (demo_state == 4) {


            if (arm==0) {  // Due to oddity in custom end effector fabrication
                // right gripper has inverted action release<=>grip

                if (gripper_open) {
                    state1_init();
                    demo_state =1;


                }
                } else {

                if (gripper_closed) {
                    state1_init();
                    demo_state =1;


                }

            }

        }
```

```
        /////

        if (demo_state == 5) {
            ROS_INFO("State 5") ;

            if (arm==1) {  // Due to oddity in custom end effector fabrication
                // right gripper has inverted action release<=>grip

                if (gripper_open) {
                    state3_init();
                    demo_state =3;
                }
                } else {


                if (gripper_closed) {
                    state3_init();
                    demo_state =3;

                }
            }

        }


        rate.sleep();

    }

    //switching back to POSITION_MODE to avoid "dead arms"

    comMsg.mode = comMsg.POSITION_MODE;
    for(int i = arm_i; i < arm_i+7 ; i++){ //arm_i is set to 2 or 9 depending on
which arm

        comMsg.command.push_back(lastStateMsg.position[i]) ;
        comMsg.names.push_back(lastStateMsg.name[i]) ;
    }
    if(arm){
        pubRight.publish(comMsg);

        }else if(!arm){
        pubLeft.publish(comMsg);
    }

    return 0;
}
```

APPENDIX B

LAUNCH FILE

```
<launch>
```

```
<!-- Uncomment one block or both for two-arm demo, each arm does the same action
Matthew Stein  23/07/2014 --><!-- Recognizes two positions, start and end, blind
joint-mode move to these positions, therefore these
       can be anywhere -->
<!--  Left robot control  - three nodes with "left" parameter -->
 <node pkg="grasp" type="state_control_node" name="left_drink_state" cwd="node"
respawn="false" output="log" clear_params="true" >

       <param name="arm" type="string" value = "left" />
       <param name="joint_start_0" type="string" value="-0.5829126987304688" />
       <param name="joint_start_1" type="string" value="1.538199233294678" />
        <param name="joint_start_2" type="string" value="0.3804272349609375" />
        <param name="joint_start_3" type="string" value="-0.003067961572265625" />
        <param name="joint_start_4" type="string" value="1.8657041311340332" />
        <param name="joint_start_5" type="string" value="-1.1263253922180176" />
        <param name="joint_start_6" type="string" value="-0.44447093278198246" />
       <param name="joint_end_0" type="string" value="-0.5829126987304688" />
       <param name="joint_end_1" type="string" value="1.538199233294678" />
        <param name="joint_end_2" type="string" value="0.3804272349609375" />
        <param name="joint_end_3" type="string" value="-0.003067961572265625" />
        <param name="joint_end_4" type="string" value="1.8657041311340332" />
        <param name="joint_end_5" type="string" value="-1.1263253922180176" />
        <param name="joint_end_6" type="string" value="-0.44447093278198246" />

    </node>

    <node pkg="grasp" type="grasp_node" name="left_drink_control" cwd="node"
respawn="false" output="log" clear_params="true" >
       <param name="arm" type="string" value = "left" />

    </node>


    <node pkg="ar_pose" type="ar_single" name="left_pose" output="log" >

       <remap from="/camera/image_raw" to ="/cameras/left_hand_camera/image" />
        <remap from="/camera/camera_info" to
="/cameras/left_hand_camera/camera_info" />
       <param name="marker_width" type="double" value = "20.0" />
        <param name="marker_frame" type="string" value = "left_ar_marker"/>
       <param name="threshold" type="double" value = "70" />
        <param name="use_history" type="bool" value = "true" />

    </node>

<!--  Right robot control  - three nodes with "right" parameter  -->
 <node pkg="grasp" type="state_control_node" name="right_drink_state" cwd="node"
respawn="false" output="log" clear_params="true" >

       <param name="arm" type="string" value = "right" />
       <param name="joint_start_0" type="string" value="0.6308495982971192" />
       <param name="joint_start_1" type="string" value="1.219514724975586" />
        <param name="joint_start_2" type="string" value="-0.6043884297363281" />
        <param name="joint_start_3" type="string" value="-0.19903400700073243" />
        <param name="joint_start_4" type="string" value="-2.6890683180908206" />
        <param name="joint_start_5" type="string" value="-1.4131797992248536" />
```

```
        <param name="joint_start_6" type="string" value="0.7915340856445313" />
       <param name="joint_end_0" type="string" value="0.8682331249511719" />
       <param name="joint_end_1" type="string" value="1.8829614149780274" />
        <param name="joint_end_2" type="string" value="-0.3861796629089356" />
        <param name="joint_end_3" type="string" value="-0.23546605067138673" />
        <param name="joint_end_4" type="string" value="-2.230791558233643" />
        <param name="joint_end_5" type="string" value="-0.7761942777832032" />
        <param name="joint_end_6" type="string" value=" 0.4345000576721192" />


    </node>

 <node pkg="grasp" type="grasp_node" name="right_drink_control" cwd="node"
respawn="false" output="log" clear_params="true" >
       <param name="arm" type="string" value = "right" />

    </node>


    <node pkg="ar_pose" type="ar_single" name="right_pose" output="log" >

       <remap from="/camera/image_raw" to ="/cameras/right_hand_camera/image" />
        <remap from="/camera/camera_info" to
="/cameras/right_hand_camera/camera_info" />
       <param name="marker_width" type="double" value = "20.0" />
        <param name="marker_frame" type="string" value = "right_ar_marker"/>
       <param name="threshold" type="double" value = "70" />
        <param name="use_history" type="bool" value = "true" />

    </node>

</launch>
```