Trent Ziemer

Dr. Newman

EECS 397

October 3, 2016

Homework 4: Three Link Robot Arm Control

I added a second joint with the same properties as the first joint and added a third arm link with a smaller mass than the second arm link but otherwise identical. The third link mass was 0.4 kg instead of 1 kg. This change was made because the first joint controller had too much difficulty moving the first link with double the weight and significantly more rotational inertia added onto it's load.

The new controller simply duplicated every single thing done for the joint 1 as for the joint 2, except for node handle initializations, and the Gazebo services. For example, an old variable name might be lorem\_ipsum, and the new variable for joint 2 would be lorem\_ipsum2 or lorem2\_ipsum.

The sine commander node simple computes a hard-coded sine wave signal and publishes this to the controller node (on pos\_cmd and pos\_cmd2 for each joint, respectively). The equations are updated by 0.0001 100 times per second, and are shown below:

pos\_cmd.data = 1\*sin(5\*input\_float.data);

pos\_cmd2.data = 1\*sin(20\*input\_float.data);

The video shows the (very poorly tuned) controller moving the second link about 4 times as slow as the first, about a radian each direction. There is normal gravity.

Source Code Copies

This is the XML code for the new description of the robot arm:

<?xml version="1.0"?>

<robot name="one\_DOF\_robot">

<!-- Used for fixing robot to Gazebo 'base\_link' -->

<link name="world"/>

<joint name="glue\_robot\_to\_world" type="fixed">

<parent link="world"/>

<child link="link1"/>

</joint>

<!-- Base Link -->

<link name="link1">

<collision>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<geometry>

<box size="0.2 0.2 0.7"/>

</geometry>

</collision>

<visual>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<geometry>

<box size="0.2 0.2 1"/>

</geometry>

</visual>

<inertial>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<mass value="1"/>

<inertia

ixx="1.0" ixy="0.0" ixz="0.0"

iyy="1.0" iyz="0.0"

izz="1.0"/>

</inertial>

</link>

<!-- Moveable Link #1 -->

<link name="link2">

<collision>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<geometry>

<cylinder length="1" radius="0.1"/>

<!--box size="0.15 0.15 0.8"-->

</geometry>

</collision>

<visual>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<geometry>

<cylinder length="1" radius="0.1"/>

</geometry>

</visual>

<inertial>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<mass value="1"/>

<inertia

ixx="0.1" ixy="0.0" ixz="0.0"

iyy="0.1" iyz="0.0"

izz="0.005"/>

</inertial>

</link>

<!-- Joint 1 -->

<joint name="joint1" type="continuous">

<parent link="link1"/>

<child link="link2"/>

<origin xyz="0 0 1" rpy="0 0 0"/>

<axis xyz="0 1 0"/>

</joint>

<!-- Moveable Link # -->

<link name="link3">

<collision>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<geometry>

<cylinder length="1" radius="0.1"/>

<!--box size="0.15 0.15 0.8"-->

</geometry>

</collision>

<visual>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<geometry>

<cylinder length="1" radius="0.1"/>

</geometry>

</visual>

<inertial>

<origin xyz="0 0 0.5" rpy="0 0 0"/>

<mass value="0.4"/>

<inertia

ixx="0.1" ixy="0.0" ixz="0.0"

iyy="0.1" iyz="0.0"

izz="0.005"/>

</inertial>

</link>

<!-- Moveable Link #2 -->

<joint name="joint2" type="continuous">

<parent link="link2"/>

<child link="link3"/>

<origin xyz="0 0 1" rpy="0 0 0"/>

<axis xyz="0 1 0"/>

</joint>

</robot>

This is the code for the new minimal\_joint\_controller:

// MINIMAL JOINT CONTROLLER for HW4 of EECS 397 ROS CLASS COPIED FROM EXAMPLE CODE - TRENT ZIEMER 10/3/2016

#include <ros/ros.h> //ALWAYS need to include this

#include <gazebo\_msgs/GetModelState.h>

#include <gazebo\_msgs/ApplyJointEffort.h>

#include <gazebo\_msgs/GetJointProperties.h>

#include <sensor\_msgs/JointState.h>

#include <string.h>

#include <stdio.h>

#include <std\_msgs/Float64.h>

#include <math.h>

#include <stdio.h>

//a simple saturation function; provide saturation threshold, sat\_val, and arg to be saturated, val

double sat(double val, double sat\_val) {

if (val>sat\_val)

return (sat\_val);

if (val< -sat\_val)

return (-sat\_val);

return val;

}

double g\_pos\_cmd=0.0; //position command input-- global var

void posCmdCB(const std\_msgs::Float64& pos\_cmd\_msg)

{

ROS\_INFO("received value of pos\_cmd is: %f",pos\_cmd\_msg.data);

g\_pos\_cmd = pos\_cmd\_msg.data;

}

// A copy of the posCmdCB function (above) modified to operate on the second

// global position command instead of the g\_pos\_cmd variable.

double g\_pos\_cmd2=0.0;

void posCmdCB2(const std\_msgs::Float64& pos\_cmd\_msg2)

{

ROS\_INFO("received value of pos\_cmd2 is: %f",pos\_cmd\_msg2.data);

g\_pos\_cmd2 = pos\_cmd\_msg2.data;

}

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

ros::init(argc, argv, "minimal\_joint\_controller");

ros::NodeHandle nh;

ros::Duration half\_sec(0.5);

// make sure service is available before attempting to proceed, else node will crash

bool service\_ready = false;

while (!service\_ready) {

service\_ready = ros::service::exists("/gazebo/apply\_joint\_effort",true);

ROS\_INFO("waiting for apply\_joint\_effort service");

half\_sec.sleep();

}

ROS\_INFO("apply\_joint\_effort service exists");

ros::ServiceClient set\_trq\_client =

nh.serviceClient<gazebo\_msgs::ApplyJointEffort>("/gazebo/apply\_joint\_effort");

service\_ready = false;

while (!service\_ready) {

service\_ready = ros::service::exists("/gazebo/get\_joint\_properties",true);

ROS\_INFO("waiting for /gazebo/get\_joint\_properties service");

half\_sec.sleep();

}

ROS\_INFO("/gazebo/get\_joint\_properties service exists");

ros::ServiceClient get\_jnt\_state\_client =

nh.serviceClient<gazebo\_msgs::GetJointProperties>("/gazebo/get\_joint\_properties");

gazebo\_msgs::ApplyJointEffort effort\_cmd\_srv\_msg;

gazebo\_msgs::GetJointProperties get\_joint\_state\_srv\_msg;

// Begin modifications made for two joints

// Gazebo service message objects for joint 2

gazebo\_msgs::ApplyJointEffort effort\_cmd\_srv\_msg2;

gazebo\_msgs::GetJointProperties get\_joint\_state\_srv\_msg2;

ros::Publisher trq\_publisher = nh.advertise<std\_msgs::Float64>("jnt\_trq", 1);

ros::Publisher vel\_publisher = nh.advertise<std\_msgs::Float64>("jnt\_vel", 1);

ros::Publisher pos\_publisher = nh.advertise<std\_msgs::Float64>("jnt\_pos", 1);

ros::Publisher joint\_state\_publisher = nh.advertise<sensor\_msgs::JointState>("joint\_states", 1);

// Published data for the state of joint 2

ros::Publisher trq\_publisher2 = nh.advertise<std\_msgs::Float64>("jnt\_trq2", 1);

ros::Publisher vel\_publisher2 = nh.advertise<std\_msgs::Float64>("jnt\_vel2", 1);

ros::Publisher pos\_publisher2 = nh.advertise<std\_msgs::Float64>("jnt\_pos2", 1);

ros::Publisher joint\_state\_publisher2 = nh.advertise<sensor\_msgs::JointState>("joint\_states2", 1);

ros::Subscriber pos\_cmd\_subscriber = nh.subscribe("pos\_cmd",1,posCmdCB);

// Write to this topic - pos\_cmd2 - to send a command to joint 2

ros::Subscriber pos\_cmd\_subscribe2r = nh.subscribe("pos\_cmd2",1,posCmdCB2);

std\_msgs::Float64 trq\_msg;

std\_msgs::Float64 q1\_msg,q1dot\_msg;

sensor\_msgs::JointState joint\_state\_msg;

// Data for the state of joint 2 variables

std\_msgs::Float64 trq\_msg2;

std\_msgs::Float64 q2\_msg,q2dot\_msg;

sensor\_msgs::JointState joint\_state\_msg2;

double q1, q1dot;

// State variables for joint 2

double q2, q2dot;

double dt = 0.01;

ros::Duration duration(dt);

ros::Rate rate\_timer(1/dt);

effort\_cmd\_srv\_msg.request.joint\_name = "joint1";

effort\_cmd\_srv\_msg.request.effort = 0.0;

effort\_cmd\_srv\_msg.request.duration= duration;

get\_joint\_state\_srv\_msg.request.joint\_name = "joint1";

// Set some message information for joint 2

effort\_cmd\_srv\_msg2.request.joint\_name = "joint2";

effort\_cmd\_srv\_msg2.request.effort = 0.0;

effort\_cmd\_srv\_msg2.request.duration= duration;

get\_joint\_state\_srv\_msg2.request.joint\_name = "joint2";

//double q1\_des = 1.0;

double q1\_err;

double Kp = 50.0;

double Kv = 6;

double trq\_cmd;

// Controller variables for the joint 2 controller

// We assume two different controllers for each joint

double q2\_err;

double Kp2 = 50.0;

double Kv2 = 6;

double trq\_cmd2;

// set up the joint\_state\_msg fields to define a single joint,

// called joint1, and initial position and vel values of 0

joint\_state\_msg.header.stamp = ros::Time::now();

joint\_state\_msg.name.push\_back("joint1");

joint\_state\_msg.position.push\_back(0.0);

joint\_state\_msg.velocity.push\_back(0.0);

// Do the same for joint 2

joint\_state\_msg2.header.stamp = ros::Time::now();

joint\_state\_msg2.name.push\_back("joint2");

joint\_state\_msg2.position.push\_back(0.0);

joint\_state\_msg2.velocity.push\_back(0.0);

// Commanding main loop

while(ros::ok()) {

get\_jnt\_state\_client.call(get\_joint\_state\_srv\_msg);

q1 = get\_joint\_state\_srv\_msg.response.position[0];

q1\_msg.data = q1;

pos\_publisher.publish(q1\_msg);

q1dot = get\_joint\_state\_srv\_msg.response.rate[0];

q1dot\_msg.data = q1dot;

vel\_publisher.publish(q1dot\_msg);

joint\_state\_msg.header.stamp = ros::Time::now();

joint\_state\_msg.position[0] = q1;

joint\_state\_msg.velocity[0] = q1dot;

joint\_state\_publisher.publish(joint\_state\_msg);

//ROS\_INFO("q1 = %f; q1dot = %f",q1,q1dot);

//watch for periodicity

q1\_err= g\_pos\_cmd-q1;

if (q1\_err>M\_PI) {

q1\_err -= 2\*M\_PI;

}

if (q1\_err< -M\_PI) {

q1\_err += 2\*M\_PI;

}

trq\_cmd = Kp\*(q1\_err)-Kv\*q1dot;

//trq\_cmd = sat(trq\_cmd, 10.0); //saturate at 1 N-m

trq\_msg.data = trq\_cmd;

trq\_publisher.publish(trq\_msg);

// send torque command to Gazebo

effort\_cmd\_srv\_msg.request.effort = trq\_cmd;

set\_trq\_client.call(effort\_cmd\_srv\_msg);

//make sure service call was successful

bool result = effort\_cmd\_srv\_msg.response.success;

if (!result)

ROS\_WARN("service call to apply\_joint\_effort failed!");

// Do everything we did above for joint 1 as for joint 2, below:

get\_jnt\_state\_client.call(get\_joint\_state\_srv\_msg2);

q2 = get\_joint\_state\_srv\_msg2.response.position[0];

q2\_msg.data = q2;

pos\_publisher2.publish(q2\_msg);

q2dot = get\_joint\_state\_srv\_msg2.response.rate[0];

q2dot\_msg.data = q2dot;

vel\_publisher2.publish(q2dot\_msg);

joint\_state\_msg2.header.stamp = ros::Time::now();

joint\_state\_msg2.position[0] = q2;

joint\_state\_msg2.velocity[0] = q2dot;

joint\_state\_publisher2.publish(joint\_state\_msg2);

//ROS\_INFO("q1 = %f; q1dot = %f",q1,q1dot);

//watch for periodicity

q2\_err= g\_pos\_cmd2-q2;

if (q2\_err>M\_PI) {

q2\_err -= 2\*M\_PI;

}

if (q2\_err< -M\_PI) {

q2\_err += 2\*M\_PI;

}

trq\_cmd2 = Kp2\*(q2\_err)-Kv2\*q2dot;

//trq\_cmd = sat(trq\_cmd, 10.0); //saturate at 1 N-m

trq\_msg2.data = trq\_cmd2;

trq\_publisher2.publish(trq\_msg2);

// send torque command to Gazebo

effort\_cmd\_srv\_msg2.request.effort = trq\_cmd2;

set\_trq\_client.call(effort\_cmd\_srv\_msg2);

//make sure service call was successful

bool result2 = effort\_cmd\_srv\_msg2.response.success;

std::cout << trq\_cmd << " and " << trq\_cmd2 << std::endl;

if (!result2)

ROS\_WARN("service call to apply\_joint\_effort failed!");

ros::spinOnce();

rate\_timer.sleep();

}

}

This is the code for the sine\_commander:

//NODE TO SEND OUT SINUSOIDAL POS COMMANDS BLINDLY FOR HW4 EECS 397 ROS - TRENT ZIEMER 10/3/2016

// BASED OFF EXAMPLE SINE COMMANDER/MINIMAL PUBLISHER IN CLASS CODE

#include <ros/ros.h>

#include <std\_msgs/Float64.h>

// Include math for sine function computations

#include <math.h>

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

ros::init(argc, argv, "two\_joint\_sin\_command"); // name of this node will be "minimal\_publisher2"

ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS

ros::Publisher j1\_pos\_cmd = n.advertise<std\_msgs::Float64>("pos\_cmd", 1);

ros::Publisher j2\_pos\_cmd = n.advertise<std\_msgs::Float64>("pos\_cmd2", 1);

//"topic1" is the name of the topic to which we will publish

// the "1" argument says to use a buffer size of 1; could make larger, if expect network backups

std\_msgs::Float64 input\_float; //create a variable of type "Float64",

std\_msgs::Float64 pos\_cmd; //create a variable of type "Float64",

std\_msgs::Float64 pos\_cmd2; //create a variable of type "Float64",

ros::Rate naptime(100); //create a ros object from the ros “Rate” class;

//set the sleep timer for 1Hz repetition rate (arg is in units of Hz)

input\_float.data = 0.0;

// do work here in infinite loop (desired for this example), but terminate if detect ROS has faulted

while (ros::ok())

{

input\_float.data = input\_float.data + 0.0001; //increment by 0.0001 each iteration

pos\_cmd.data = 1\*sin(5\*input\_float.data);

pos\_cmd2.data = 1\*sin(20\*input\_float.data);

j1\_pos\_cmd.publish(pos\_cmd); // publish the value--of type Float64--

j2\_pos\_cmd.publish(pos\_cmd2);

naptime.sleep();

}

}