Massachusetts Institute of Technology

Robotics: Science and Systems I

Lab 7: Grasping and Object Transport

Distributed: 3/11/2015, 3pm
Due: 3/18/2015, 3pm

Objectives and Lab Overview

Your objective in this lab is to understand grasping and object transport. You will build an arm with a gripper for your robot. You will incorporate the arm in your robot. You will then use the arm to pick up objects and transport them to desired locations.

This lab will give you the technical skills to incorporate grasping and manipulation capabilities into your robot. This lab will also enhance your knowledge of the mechanics of objects in contact which is an important aspect of interfacing computation to the physical world.

Time Accounting and Self-Assessment:

Make a dated entry called "Start of Grasping Lab" on your Wiki's Self-Assessment page. Before doing any of the lab parts below, answer the following questions:

- **Programming**: How proficient are you at writing large programs in Java (as of the start of this lab)?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)
- **Hardware**: How proficient are you at modifying the hardware of your robot?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)
- **Mechanics of Manipulation**: How proficient are you at mechanics and kinematics?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)
- **Visual Navigation**: How proficient are you at using the vision and navigation software on your robot?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)

To start the lab, you should have:

- The arm/gripper kit: 12 laser-cut pieces for the arm and gripper, 3 servos, 1 bump sensor, mounting hardware.
- Your notes from the Grasping, Kinematics, and Manipulation lectures
- Arm Assembly Instructions

Physical Units

We remind you to use MKS units (meters, kilograms, seconds, radians, watts, etc.) throughout the course and this lab. In particular, this means that *whenever you state a physical quantity, you must state its units*. Also show units in your intermediate calculations.

Part 1: Building the Arm

In this part of the lab you will use the kit we give you to assemble and install an arm with gripper for your robot. The exemplar robot will be available for inspection. It shows the end result of your assembly. The arm assembly handout contains pictorial step-by-step assembly instructions.

Before assembling the arm it is a good idea to test the servos. **The shoulder servo MUST always be run with the robot powered by the battery, not the AC adapter due to its current requirements.** You can test the servos by starting Part 2 and testing the servos in parallel with some of the assembly steps.

Deliverables: Create a new page on your wiki called "Grasping Lab Report Group N." Take some pictures of your arm while being constructed, and a picture of the final result, put these on your wiki page. Please record in the difficulties you encountered, if any.

Part 2: Controlling the Arm

You should begin by adding the new lab source code to your group repository following the usual procedure. In addition, you will need to re-export `rss_msgs` and `uorc_publisher`. The RSS code base contains support for your arm servos and the additional bump sensor. The `uorc_listener` node listens to the message `/command/Arm` for arm commands and the `uorc_publisher` publishes the current arm state on `/rss/ArmStatus`.

Arm parameters

The ORC board (and your code, by extension) has the ability to support servos through (*fast* DIO ports on the ORC board). The source code you are given assumes that all of your servos are identical, and each accepts a 16 bit PWM value which is integrated by the servo electronics into a rotational position. However, each model of servo that you have been given has specific maximum and minimum angles that it can express. These correspond to maximum and minimum PWM values. You will need to add code to handle this differentiation. In addition, you will need to calibrate each servo's mapping from PWM to the corresponding angular value. Instructions for how to do this will be covered in the Arm Control subsection of the lab.

As an exercise, read through `uorc_listener` code to determine the appropriate ports for your arm servos. Note that you may elect to use different ports, so it is important that you know the location of this specification in the code.

Arm class libraries

You can view the current state of the Arm by subscribing and handling `rss_msgs.ArmMsg`. As a reminder, to subscribe to arm messages, implement a handler using the following code snipets:

```
private Subscriber<org.ros.message.rss_msgs.ArmMsg> armSub;
...
@Override public void onStart(Node node){
   armSub = node.newSubscriber("rss/ArmStatus", "rss_msgs/ArmMsg");
   armSub.addMessageListener(new ArmListener(...));
}
...
public class ArmListener implements MessageListener<ArmMsg> {
   ...
   @Override public void onNewMessage(ArmMsg msg) {
         ...
   }
}
```

The `ArmMsg` contains six floats, the first three of which correspond to the first three PWM I/O ports.

You can also subscribe to the gripper bump sensor on topic `/rss/BumpSensors` with message type `rss_msgs/BumpMsg`.

Arm control

Your goal in this part of the lab is to implement simple, reliable control of the arm. We have provided for you a helper GUI for exploring arm control, called `ArmPoseGui`.

For each arm servo, you need to determine its **maximum** and **minimum** PWM values.

The servo cannot be physically moved past these values; if you try, the command will either be ignored, or worse, the servo motor will chatter against the physical limits. You should take into account not only the range of motion of the servo itself, but also within the context of the arm's range of motion.

For each servo, use the slider in the `ArmPoseGui` to determine what the extreme PWM values are. Be very careful as the arm may move very fast when you do this.

You need to know what PWM values correspond to actual angles, in order to compute a conversion between angles and PWM ticks. For each servo, move the servo to the position that you consider to be $\theta_1 = 0$ radians using the slider in `ArmPoseGui`. Note the PWM value, call it PWM_1. Now, move the servo to some other angle, such as $\theta_2 = \pi/2$ radians. You will have to measure this angle carefully. Note this PWM value as well, call it PWM_2. You can use these two data points to compute a conversion between angles and PWM by fitting a line and interpolating for desired values. The slope of your line will be:

$$m = \frac{\theta_2 - \theta_1}{PWM_2 - PWM_1} \tag{1}$$

The theta-intercept of your line can be determined by plugging in one data point:

$$\theta_i = \theta_1 - m \cdot PWM_1 \tag{2}$$

Recognize that you'll need separate conversion factors for each servo motor, including the gripper.

Now, create a new file called `Grasping.java` in which you will place the code for this lab. Begin by writing a simple Java program that receives the arm messages. Using the appropriate conversion factors for each servo, write `handle(ArmMsg msg)`, which moves each servo through its full range of motion, moving all servos concurrently. This handler should repeat the motion indefinitely. Note that this will require implementing a (fairly simple) finite state machine inside your arm message handler.

One caveat: You should be careful about moving any servo through too large a range of motion in a single step. You might want to experiment with how large a range of motion each servo can tolerate, but a good rule of thumb is that **no servo should move more than 1 radian per iteration**. Moving faster could cause the servos to skip, fuses to blow or worse, an unexpected motion could slam the arm into the ground destroying it. This slew rate control can be accomplished by implementing a clamped feed-forward control step for each servo.

Hint: You may want to write a joint controller class and create subclasses for each of the shoulder, wrist, and gripper joints. This will help you to capture the common methods for servo control, while enabling specific behaviors for each joint.

Deliverables: Your wiki should include:

- *Your minimum and maximum PWM measurements for each servo*
- *Your angle measurements and your angle-to-PWM conversions for each arm*

Arm control and inverse kinematics

Your goal in this part of the lab is to characterize the gripper position in terms of joint angles. Notice that you have two revolute joints (the shoulder and the elbow) that control the position of the end effector. There will, in general, be two sets of solutions mapping between the joint angles and end-effector position in body coordinates. You will encounter this ambiguity in your computation, and you must choose one solution (based on continuity, servo bounds, etc).

- Measure the length of each arm segment. Note: use the distal end of the gripper as the end of your kinematic chain.
- Determine the forward kinematic equation that maps joint angles to end effector positions.
- Determine the inverse kinematic equation that maps end effector positions to joint angles.
- Choose an end effector position in the x, z plane in the robot frame. For each of several end effector positions, compute the appropriate joint angles, move the servos to those angles, and measure the position of the end effector in body coordinates.
- Place an object in the gripper, and close the gripper. (You should be able to close the gripper using a Java program. Do not force the gripper jaws closed by hand.) Repeat the measurement process with the object in the gripper.

Deliverables: Your wiki should contain a set of explicit assumptions you made in building an inverse kinematic arm controller. You should also discuss how accurate your controller is, and how you might correct it. Are there any failure modes and what are they, if any?

- *Your measurements of your arm*
- *Your mathematical model of the inverse kinematics*
- *The expected and measured end effector positions with and without an object in the grasp.*

Note: The controller for the PWM servos is feed-forward. As a result, the `uorc_publisher` only publishes the last commanded values for the servos (and not the true position of the servo). However, this information may be useful to account for time delays between the nodes running on the workstation and the nodes running on the netbook. We suggest you use the feedback from `uorc_publisher` to determine when the arm has reached a position.

Optional: The ORC board does not contain enough input lines to allow us to equip the servos with encoders and so that you could use PD controller that you implemented in earlier labs. However, you do have an additional sensor: the camera. How might you incorporate the camera to correct for arm controller errors?

Checkpoint

The staff will walk around to do a checkoff. We will be looking to see that:

- Your arm is constructed and mounted on the robot
- You can control your arm via the ArmPoseGUI
- You can control your arm via inverse kinematics

Part 3: Grasping and Transporting an Object

Arm gymnastics

In this part of the lab you will build arm behaviors. The arm control libraries can be used to program “arm gymnastics”. Write a program that controls the arm through a sequence of moves: open-gripper, close-gripper, move-up, bend-elbows, touch-the-ground. To do this you will have to calibrate the arm to differentiate between an open and closed arm, and to detect when the arm touches the ground. detect impediments. Make sure you slew the commanded servo positions (only move at most one radian per iteration), otherwise you will destroy your arm when it mistakenly hits the ground (which is not fun).

Write a program to implement:

1. open-gripper
2. close-gripper
3. move-up with a desired angle

4. bend-elbow with a desired angle
5. move-to-ground

and then demonstrate how you can sequence these behaviors as “arm gymnastics”.

Deliverables: Your lab report should show a video sequence of the arm gymnastics and an explanation for how you controlled each movement.

Grasp and Transport

In this part of the lab, your robot will pick up an object and move it a specified distance. To begin, place the arm of your robot on the floor, in an open position. Then, manually place an object (one of the colored cubes) in the gripper. This action should be detected by the bump sensor, which should then trigger a grasping behavior for the arm. Once the object is grasped, the arm should be lifted and the object should be transported some distance forward. You may choose any distance and direction for this displacement.

To complete this functionality, write software to do the following:

1. Initialize the arm and move the joints to their pre-grasping position.
2. Wait for an object to penetrate the grasp region of the gripper by monitoring the bump sensor. (Remember: the bump sensor should be connected to slow digital I/O port 2.)
3. Grasp the penetrating object.

   This part is a little tricker than simply closing the hand. You will have to calibrate your gripper for two things: (1) to decide how tight to close it around the object, and (2) to make sure that you maintain complete closure of your object so that when you lift it off the ground it will not fall out of the hand.

   If you want to check whether the object has fallen out, how will you do so? Is this a reliable method? Can you think of a more reliable one? (Hint: a different sensor)
4. Lift the grasped object off the ground.

   Your lifting method should detect and recover from error. Error occurs when your hand drops the object. Implement recovery by trying to grasp once again. The bump sensor will also give you an empty hand signal in this case.
5. Move the robot to deposit the object at the new location.
6. Place the object on the ground and move the robot back to its original starting point. Measure the error between the desired location of the object and its true placement for several trials.

You may find it helpful to begin by drawing a diagram of your finite state machine, and identifying which components of your system are active in each state.

Deliverables: Your wiki report should include a video of this task and an explanation for your implementations. Please include a discussion of your calibration parameters used for impediments (for closing the hand with and without the object) and for detecting when the object slips out of the grasp. How reliable is your control of arm gymnastics? How reliable is the control for grasping? How accurate is the displacement of the object? Discuss the failure modes of this functionality. Please also give us a pointer to the code and answers to the questions above.

Part 4: Searching For and Retrieving an Object

> *I have done this approach two hundred and thirteen times on the simulator. We are NOT where we should be.*
> — Col. Robert Iverson, The Core

Your goal in this part of the lab is to integrate the object pick-and-carry implementation from the previous section, with your visual servoing code from the Visual Servoing Lab. (We’re ecological roboticists – we recycle.) The basic idea is to visually servo to a block of a specific color and maintain an appropriate fixation distance, such that you can then retrieve and transport the object. You are free to use your own code from the Visual Servoing Lab or the solution code. The issues of colour calibration, blob centering, etc., are the same regardless of whether you use your solution or ours.

If you recall, the `BlobTracking` class contains the `apply(Image src, Image dest)` method, which extracts all the blobs of the appropriate hue from the `src` image, and highlights the blobs in the `dest` image. Your `BlobTracking` class is not calibrated to the new object, so you must first re-calibrate.

Recall from the Visual Servoing Lab that this is accomplished by holding the object you wish to calibrate within the camera's view, while outputting the HSB histogram in the `VisionGUI`. If the object you wish to track is the dominant feature in the scene, then the dominant hue in the histogram should be the hue of your object. Once you have identified the hue of your block, edit the target hue level used by your classifier (in the solution code for `VisualServo.java`, this is done by setting the `target_hue_level` parameter with `Param.set`, so that no physical changes need to be made to `BlobTracking.java`.

In `Grasping.java`, subscribe to the video messages on `/rss/video`. Instantiate a `BlobTracking` object and pass in the video. Publish some debug output on `/rss/blobVideo`, as suggested below, which shows the highlighted blobs.

```
Publisher<org.ros.message.sensor_msgs.Image> vidPub;
...
vidPub = node.newPublisher("/rss/blobVideo", "sensor_msgs/Image");
...
Image dest = ...;
...
org.ros.message.sensor_msgs.Image pubImage =
     new org.ros.message.sensor_msgs.Image();
pubImage.width = width;
pubImage.height = height;
pubImage.encoding = "rgb8";
pubImage.is_bigendian = 0;
pubImage.step = width*3;
pubImage.data = dest.toArray();
vidPub.publish(pubImage);
```

Test your blob tracker by placing your object in the field of view of the camera, and watch the display. You should see your object highlighted in the camera panel.

The next important piece is the visual servoing, which requires that you know the size of the object in the field of view to determine the appropriate stand-off: if the object appears too small, you need to drive closer, and if the object appears too large, you would need to back up. Your ability to determine the distance to the object depends on knowing how large the object is. Let us assume that the radius returned by the blob tracker is a reasonable approximation of the object width. Measure the object's width, and modify the target radius size used by your blob tracker (again, the solutions utilize the `Param` class for setting the `target_radius` parameter). Test your visual servoing code by having your robot servo to the object as you did in the Visual Servoing Lab.

The final parameter you need to calibrate is the stand-off distance. You need to determine how far the block is from the center of the camera when the block is inside the gripper. You should be able to measure this parameter directly by placing the object in the gripper.

Once you have determined that you are able to visually servo to the object, you need to co-ordinate the object pick-and-carry implementation from Part 3 with your visual servoing code. In particular, once the bump sensor detects the object, you should stop the robot translating and stop processing the visual servoing commands. At the same time, you should start closing the gripper preparatory to lifting the object.

Deliverables: Your wiki report should contain:

- *A screenshot of your block*
- *Your calibration histogram*
- *Your calibration parameters (hue, size, stand-off distance)*
- *A description of each module and algorithm in each, the APIs between the modules*
- *A description of your robot operation. How well does your visual servoing work in this lab compared to the Visual Servoing Lab? Include a video of your robot running fully autonomously, as in the previous lab part.*
- *The failure modes of this functionality*
- *The task allocations within your team*

Optional: You might consider using a few different stand-off distances and implementing your visual servoing code in the following manner:

1. Retract the arm fully, so that it is out of the field of view of the camera
2. Visually servo to the block with a stand-off distance such that you are close, but not yet gripping the block, for example, roughly .5m away.
3. Lower the arm so that the gripper is at the right height to grip the block
4. Visually servo to the block with the correct stand-off to be able to grip the block, and monitor the bump sensor
5. Start lifting once the bump sensor detects an obstacle

Why might we recommend this visual servoing method?

Wrap Up

Report the time spent on each part of the lab in person-hours and indicate what elements were done independently or in pairs, triples or as a full group.