Massachusetts Institute of Technology

Robotics: Science and Systems I

Lab 8: Grasping and Object Transport

Distributed: Wednesday, 4/1/2009, 3pm
Checkpoint: Monday, 4/6/2009, 3pm
Due: Wednesday, 4/8/2009, 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 break-beam sensor, mounting hardware.
- Your notes from the Grasping, Kinematics, and Manipulation lectures

In addition to this lab specification (Handout 8-A), you should have the following handouts:

1. Handout 8-B: *Arm Assembly Instructions*
2. Handout 8-C: *Sharp IS471F Datasheet*, available online

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. Handout 8-B contains pictorial step-by-step instructions.

Before assembling the arm it is a good idea to test the servos. You can do this by starting Part 2 and testing the servos without the rest of the arm attached. This can be done 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. Your robot is now empowered to manipulate its world. What is the name of your robot?

Part 2: Controlling the Arm with Carmen

You should begin by adding the new lab source code to your group repository following the usual procedure. Carmen contains support for your arm servos and the breakbeam sensor within `orc_daemon`, and from the Java class libraries in the `Arm` and `ArmMessage` classes, as well as the `ArmHandler` interface. These enable you to set (or get) the angular positions of the arm, and get the state of the break-beam sensor. It will be helpful to review the Carmen API for these classes before beginning the lab.

Arm parameters

The ORC board (and Carmen, by extension) has the ability support 4 servos (input ports 0-3 on the ORC board). In the source code you are given, Carmen 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 (in your own software, not CARMEN) to handle this differentation. 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.

Arm class libraries

You can view the current state of the `Arm` by subscribing and handling `ArmMessage`'s or by querying the `Arm` class using the static method: `public ArmMessage Carmen.Arm.query();`

To subscribe to the `ArmMessage` class, implement the `ArmHandler` interface, and the appropriate handler:

```
  public void handle (ArmMessage message);
```

The `ArmMessage` contains more data than just the arm angles, and the full message format is:

```
public class ArmMessage {
  public double joint_angles[];
  public int num_joints;
  public double joint_currents[];
  public int num_currents;
  public int gripper_closed;
  public double joint_angular_vels[];
  public int num_vels;
  public int flags;
  public double timestamp;
  public char host[];
}
```

The `joint_angles[]` field contains the current PWM values of all your servo motors, and the length matches both the `num_joints` field and also the `arm_num_joints` parameter in `carmen.ini`.

In addition the servo PWM values, you have the ability to sense the current in **2** of your servo motors. By sensing the current loads, you will be able to determine whether or not your motor has stalled. The ORC board only supports current readings for two motors, so the `joint_currents[]` field contains at most two doubles, which represent the current of servos 0 and 1 in amps. If you have servos in both ports, `joint_currents[]` will contain 2 doubles and `num_currents` will equal 2. If only one port is occupied, `num_joints` will be 1, in which case `joint_currents[]` will have length 1 and `num_currents` will have value 1.

We recommend that you put the manipulator gripper and shoulder on the servo ports that have current sensing.

The `gripper_closed` field has value 0 when the break-beam sensor is clear or value 1 when the break-beam sensor is obstructed by an object. The break-beam sensor plugs into port 12 of the ORC board.

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 the following quantities:

- `MAX_PWM`
- `MIN_PWM`

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. Setting the PWM value to zero will disable a servo. If you happen to exit the program without setting the values to zero your arm may still be trying to hold a position. You can use the class `ClearArm` to reset the arm. Run it as `java Grasping.ClearArm` to reset the arm. This is a useful command when you are developing your code.

You also 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 implements `ArmHandler`. Using the appropriate conversion factors for each servo, write `handle(ArmMessage 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.

Remember you can run `java Grasping.ClearArm` to reset the arm.

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.*

Optional: You might notice that the PWM controller is a feed-forward controller, as opposed to a feed-back controller. 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: Monday, April 7

The staff will walk around at BEGINNING of lab 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 use the Carmen library to 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. You can do this by using current as a way to 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 you 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 break-beam 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. Servo the gripper to a open position where the break-beam sensor has a clear field of view.
2. Wait for an object to penetrate the grasp region of the gripper by monitoring the break-beam sensor.
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. The only sensory feedback you have access to for this calibration is current. You will have to determine the amount of current necessary to close the grasp. Make sure that when you grasp an object the gripper fingers do not touch. This is important because you have to be able to differentiate between impediments caused by tightly grasping an object and the impediment encountered when an empty hand closes and the fingers touch. You will be able to check that an object has fallen out of the grasp also by looking at the current.
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 break-beam 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.

Hint 1: The break-beam sensor will work best at a static pose, with the gripper partially open. As the gripper changes pose, the orientation of the sensors will change and you will likely experience false-positives.

Hint 2: We have provided the utility classes `SensorTimeAverage.java` and `SensorTimeThreshold.java`. You may find these handy for filtering out sensor transients and stabilizing the perceptual states of your grasping FSM.

Hint 3: Use the servo current in the gripper instead of the break-beam sensor to discriminate when you are actually gripping the cube. The changing angle of the break-beam may yield it insufficient.

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

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 neet to be made to `BlobTracking.java`. You can alternatively set these parameters in your local `carmen.ini` file, or using `param_edit` to change the live settings). Test your blob tracker by placing your object in the field of view of the camera, and watch the display in `VisionGUI`. 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 break-beam. You should be able to measure this parameter directly by placing the object in the break-beam.

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 break-beam 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 break-beam sensor
5. Start lifting once the break-beam 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.