OBJECTIVE. The objective of this lab is to develop an understanding of forward and inverse kinematics, and to demonstrate these methods on the arm. Also, you will gain familiarity with Lightweight Communication and Marshalling (LCM) for interacting with modular code.

DELIVERABLES. This lab contains a number of TASKS that require your response. In your lab write up, you should restate the question (you can copy and paste from this document) and then provide your answer. In short, your writeup should be comprehensible without needing to reference this document.

CHECKPOINT. To keep you on track, please demonstrate your system completing the checkpoint by the date on the course calendar.

SUBMITTING YOUR WRITEUP. The writeup for all of the ArmLab parts is due at the conclusion of the arm lab; please append your writeup for part 3 to your existing part 1 & 2 document.
Forward Kinematics

Forward kinematics is the task of computing the position of your system (arm) given the joint angles.

**TASK 23.** You must run arlstaff.arm.ArmDriver to interact with the arm. This program will output LCM messages on the channel named “ARM_STATUS”. Produce a Vis visualization that subscribes to the ARM_STATUS messages and shows, in real time, a geometrically accurate depiction of the arm. You will need to measure the geometry of the arm: the accuracy of your model will ultimately affect the quality of your motion planning.

Note: you must implement a conspicuous graphical indicator of error states.

Show a screen shot of your arm, and describe how you verified that your dimensions were correct. (Hint: can you identify specific locations on the board whose locations are well known?) How did you ensure that error states are visually obvious?

**TASK 24.** Add six sliders using an april.util.ParameterGUI which control each of the six degrees of freedom of the arm. When the parameters change, construct an LCM arm_command_list_t message and publish it so that the arm moves in response to the sliders.

Before you ever send a command to the arm, consider A) what you expect the arm to do and B) what you will do if the arm does something else (potentially endangering you or the arm). Note that hitting Control-C is not a good strategy. (Explain why.) What "oh no!" strategy did you decide on?

Note: Pass “-h” to arlstaff.arm.ArmDriver in order to see the application help for how to select the device or simulated arm. We strongly recommend learning how to use this driver in simulation mode before you start moving the real arm. The simulator is not accurate on a physics simulation level, but will be close with regards to final position. However, in robotics it is usually a good idea to spend a “little” bit of time in simulation as crashed processes are much cheaper than crashed robots.

Show a screen shot of your GUI with sliders. Identify the maximum and minimum values that should ever be assigned to each servo.

**TASK 25.** Modify your Arm code to publish messages through a function that enforces these angle limits. What types of bad cases (e.g., collisions) will this not detect?

Inverse Kinematics

Our arm is quite simple in terms of geometry: given an (x,y,z) position (in arm coordinates), it is relatively easy to compute a set of joint angles that will achieve that position.

**TASK 26.** Sketch a case where there are multiple solutions. Sketch a case where there exactly one solution.

**TASK 27.** Let's handle multiple solutions by preferring situations where the elbow is UP and, if possible, the wrist points straight down. (This strategy minimizes the possibility that, when picking up one object, you'll inadvertently hit another.)

Describe your procedure for computing the inverse kinematics. (E.g, give equations and illustrate with
a drawing.)

**TASK 28.** Implement your inverse kinematics. First, implement a ParameterGUI which allows you to pick the z coordinate of your target location. Second, implement a VisEventHandler that listens to mousePresses/MouseDrags in order to compute the X and Y coordinate. (See GRay3D.intersectPlaneXY). When the user clicks, the arm should travel to that location. (Before actually trying this, read TASK 29.)

**TASK 29.** Suppose your arm is at (.1, .1, .1) and you command it to (.25, .25, .1). You compute the joint angles for the destination position and command the servo. Comment on the path (without trying it) that the servo will take. Specifically, will the arm's end effector stay at z=.1 as it moves? Explain your answer. If the answer is no, explain how this affects your motion planning.

**TASK 30.** Work through this pen and paper exercise on gradient descent (GD). Consider (optionally) implementing on your arm to improve the motion planning abilities. This problem is in 2D, but you should be able to easily modify the problem to fit the 3D arm.

Suppose the 2D arm is configured as shown below, with the red areas marked by obstacles. Your goal is to control the arm from its current position to a configuration such that the end effector is at the green circle. The end effector is shown at (11, 13), determine the desired position after one step (step size = 0.5 cm) using the gradient descent method described in class. Use the following two simple cost functions to create your 2D cost surface $\text{cost}(x, y) = 0.6 \times f(x, y) + 0.8 \times g(x, y)$

- Distance to goal: $f(x,y) = \text{obstacleFreeDistance}((x, y), \text{goal}) / \text{MAX\_DIST}$
- Obstacle avoidance: $g(x,y) = \exp(-\text{obstacleFreeDistance}((x,y), (a,b))^2 / 2^2)$

// (a,b) is location of closest obstacle to (x,y)
// Note: obstacleFreeDistance means that you cannot use the standard Euclidean distance as the cost does not transmit through obstacles. Manhattan distance, a.k.a. 4-connected, and its 8-connected variant are common approximations to Euclidean distance because of their easy connection to obstacleFreeDistance. MAX\_DIST = 32 cm. All units are in cm.

![Diagram of 2D arm configuration with obstacles and end effector](image)

Additional Notes: There are two different spaces in which we can plan: end-effector space and configuration space. When planning in end-effector space, the algorithm plans the (x, y) position of the
end-effector in the physical world. This is in contrast to planning in configuration space (a.k.a. servo-angle space) where the plans are in the N-dimensional space corresponding to the angles of the N servos. Can you foresee any problems with the GD method as described above (end-effector space)? Does the cost function take into account self collisions or other joint (non end-effector) collisions with walls? How could you modify the algorithm to handle these cases.

**CHECKOFF.** Demonstrate your solution to TASK 29.