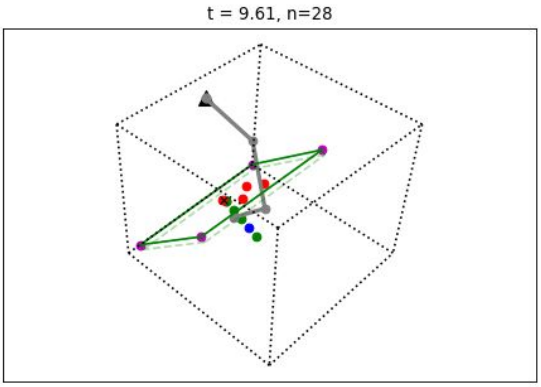
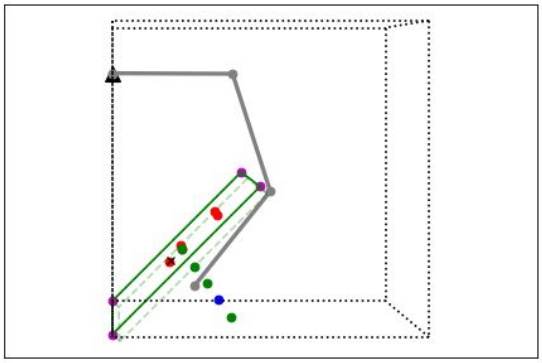
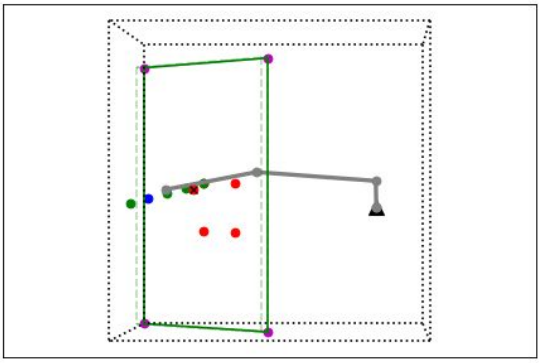
EECS 464: Hands-on Robotics - Project 2 Howto

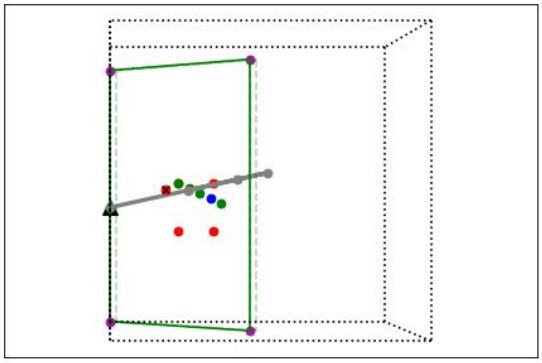
Cyan Team: Vaibhav Bafna, David Chang, Eric Wiener

Winter 2020









Overview

Our code attempts to simulate a serial manipulator robot composed only of revolute joints whose goal is to autonomously draw a square with a pen on a piece of paper. The robot has the pen attached to the end of the last segment of the robot and can be referred to as the end effector or tooltip.

We broke down our process into the following steps:

1. Set up our robot arm and workspace
2. Set up the square on the paper in the world coordinates.
3. Calibrate our robot using different points on the surface of the paper.
4. Execute a move plan that uses forward and inverse kinematics to move our robot to the desired positions of the square.

Set up our robot arm and workspace

We first defined the placement of the workspace relative to the robot arm by using the transformation matrix Tws2w, which can be used to transform from workspace to world coordinates. The first three columns of this matrix define the orientation for the x, y and z axis, while, while the fourth column defines the translation that will be applied to the workspace. An example of the Tws2w matrix is shown below.

```
Tws2w = asarray([
     [1,0,0, 0],
     [0,1,0, -5],
     [0,0,1,-10],
     [0,0,0,  1]
])
```

We defined the lengths of our arm segments and the orientations of our joints using the matrix armSpec located in myarmsim.py. Within armSpec, the number of rows denotes the arm segments. Each row describes an arm segment. The first three elements in the row describe the axes of rotation, the fourth element describes the length, and the fifth element describes the initial angle at which the segment is configured. The example below illustrates the aforementioned configuration.

```
armSpec = asarray([
  [0,0.02,1,5,0],
  [0,1,0,5,1.57],
  [0,1,0,5,0],
]).T
```

Set up the square on the paper in world coordinates

We are given the X,Y center coordinates of the square and the scale of the square in paper coordinates. We create a 4 x 4 matrix representing the square in paper coordinates, and we

transform this matrix into world coordinates so we can access the square in the workspace properly.

Move Plan

To move our pen to where we want to go, we need to set the joints to the correct angles. We rely on forward (getting end effector position given joint angles) and inverse kinematics (getting joint angles given end effector position) to accomplish this.

In order to solve the inverse kinematics of moving the arm, we use the Python package tinyik (which can be installed through pip). Using tinyik.Actuator, we created another arm model called moveArm that can be used to obtain joint angles when given an end effector position, and vice versa. We configure this arm to the same specifications defined in armspec. When we initially start the Move Plan, we calibrate moveArm using the syncArm method of the Move Plan class.

In syncArm, we get the initial joint angles of the arm by iterating through the motors and calling get_goal to obtain the initial angles of the actual arm. We then call getTool (from the Arm class) on an Arm object called idealArm (we couldn't call getTool on the actual arm), which we use to estimate the position of the real arm. This Arm object allowed us to model how an ideal arm without gravity sag and other real life properties. We plug in the angles obtained from syncArm into getTool to get an estimate of the current position of the arm. This current position can also be thought of as our starting position.

We then take one of the corner points of the square and set that as the position we want to move towards. To move to that corner point, we use np.linspace to generate an evenly distributed series of points between our current position and the goal position. We then iteratively move to each of these points. We do this to reduce possible errors that may occur from taking a large step and to try to move in a straight line rather than in an arc. At each step along the series of points, we calculate the joint angle configuration required to move the end effector to that point. We then call set_pos on the motors and plug in those angles to move the motors into those angle configurations. This is the part that actually moves the real arm and causes you to see a change in animation on your screen. We yield for a duration of 3-5 seconds each step we take. The number of steps created by linspace and the duration can be adjusted to improve performance.

The robot is controlled using the "W", "E", "R", and "T" keys, where each key corresponds to a corner of the square. When a user presses a certain key, the arm will move autonomously to the respective corner of the square. The user is updated through console logs of the progress in completing each step. Once a step is marked as complete, the user then presses to move to the next corner.

Calibration

We tried setting up a calibration procedure to determine the error in our arm's actual positions that it ends up at relative to the desired positions we want to move to. Using this error, we obtain an angle offset that can be applied when moving the motors using set_pos. The calibration steps are as follows:

1. Create a grid of evenly spaced calibration points on the paper. Recommended is 2 by 2 grid which sets 4 calibration points at the corners of the paper, but can be adjusted as you want.
2. Move to one of the grid points using the move plan. Store the angles for the position the arm ends up at. This is activated by pressing the "K" key.
3. Manually adjust the arm so the end effector is located at the grid point. Use the keys "asdf…" and "zxcv…" (more keys depending on how many joints you have) to adjust.
4. Once you are at the correct location, determine the joint angles corresponding to this location and subtract from the joint angles obtained in step 2 to calculate the angle error for each joint. Store these error values in a matrix that collects angle errors.
5. Repeat steps 2 - 4 until you've done this for all grid points. You now have a complete angle error matrix which we can call our calibration matrix.
6. Apply a linear interpolation between the grid points and the associated angle errors that can be used to map between the two values and output an angle offset. You can input in a goal position and it will output the angle offset.
7. Execute the move plan on the square corners, but now applying an angle offset each time. This angle offset will adjust the angles which get plugged into set_pos to move the motors.