Megan Sindelar

ME 449 Capstone 2022

README

The goal of this software is to generate and follow a trajectory, using feedforward and PI control, for a mobile robot with an attached 5-joint arm to pick and place a block. To look at this code, it is stored in the sub-directory called code from the main directory called Sindelar_Megan_capstone. The python file is called FinalProject.py. Within this code sub-directory, there are also three other sub-directories, each with the three separate milestone sections of code. I included this here as a helpful way to see each individual milestone.

This code works by first generating the desired trajectory. I generate this specific trajectory in 8 steps, using a mix of Cartesian and Screw Trajectory functions from the modern robotics library, and concatenate them all to store and reference. Next, I then give the robot initial conditions, based on joint limits that I implemented, as discussed later. Using these initial conditions and the desired trajectory, I compute a twist command based on feedback control, a function in my code called ‘FeedbackControl’. This function takes in the current position and configuration of the robot, calculated from the initial conditions using forward kinematics, the desired configuration, next step desired configuration, and a time step. It also takes in a 6x6 proportional gain matrix, as well as a 6x6 integral gain matrix. I set these two matrices based on previous knowledge of how PI control works, along with guess-and-check based on the performance of the program.

The ‘FeedbackControl’ function works by finding the configuration error to see how much difference there is between the current robot configuration and the desired trajectory configuration. Then, it computes the desired twist from the current desired trajectory configuration to the next desired configuration. Using the error, the new desired twist, and the proportional and gain terms, I compute the new commanding twist to send to the robot. I also output the error from this function to store so that I can plot the configuration error over time. Using the commanded twist output, I find the end-effector commanded twist control, consisting of wheel and arm speed commands. I do this by finding the pseudoinverse end-effector Jacobian of the robot and matrix multiplying it by the commanding twist from the feedback control.

Next, I test the joint limits to see if these new control inputs will command the robot to prohibited joint angles. I set these joint angles based on the configuration of the robot in the ‘testJointLimits’ function. I used Scene 3 in CoppeliaSim to see which joint angles correspond to what movement and based on my known trajectory decided to implement upper and lower bounded joint limits on joints 1, 2, 3, and 4. I will describe the method of implementing the joint limits here, and will go into more detail later on decisions when I discuss the results. To test the joint limits, I compute the next state configuration, using the ‘NextState’ function described below, based on the computed end-effector twist from the feedback control. I then check to see if any of the specified joints, meaning whichever joints I chose to impose limits on, are outside of the desired joint bounds. If they are, then I set the entire column of the corresponding Jacobian column to zeros, so that the new calculated end-effector twist, computed from the pseudoinverse, will not request any motion from that joint. This will cause the joint not to move and stay within its joint limit bounds. I then recompute the next state configuration and test again. I loop through this 10 times before setting an actual Jacobian to use for the next state configuration calculation. I chose 10 cycles because it allows me to check if any other joints go past their limits after setting a violated joint jacobian column to zeros. However, I cut the loop off at 10 because otherwise it could run forever.

After computing the new end-effector twist from the 'testJointLimits' function, to set as new control velocities, I pass it into the 'NextState' function to compute the new robot configuration to command. This function takes in the current configuration, the speed controls, the time step, maximum allowed speed, and the gripper state. This function works by first checking if the commanded control speeds are below or equal to the maximum allowable speed. If not, the function sets that speed to be equal to the max speed. Then, it computes the joint angles and wheel configurations based on Euler integration. For the joint angles, I multiply the control speed of that joint by the time step and then add it onto the current joint angle. The wheel configurations are computed the same. For the chassis configuration, I first find the body twist by multiplying the pseudoinverse of H(0) by the wheel control speeds. Then, I computed a configuration array of w_bz, v_bx, and v_by, based on whether or not w_bz is 0. I then compute the new chassis configuration based on Euler integration, which is the same process as with the joint angles. I then concatenate the new chassis, joint, and wheel configurations, along with the gripper state, to get my new commanded robot actual trajectory configuration. I store the variable and loop through this entire process for the length of the entire desired trajectory. To see this process in simulation, I save my configurations to a csv file. The csv file is named respectively, depending on the test. The best case csv is named robot_config_best.csv. Accordingly, the overshoot case csv is named robot_config_overshoot.csv, and the newTask case is named robot_config_newTask.csv.

As for my results, overall it looked pretty good, where for each case I always got back to no steady state error. Before I added PI control, my robot exhibited weird and unrealistic behavior and did not complete the task. But, once I added and tuned my PI control, it performed very well. For my best case, I tuned my PI control so that the robot picks up the block and sets it down in the correct end position. For this best case, I have a case with joint limits and one without joint limits. For the case with no joint limits, when the robot is going to pick up the block, it chooses the path where the arm travels through the body to get to the block. In real life, this isn't possible. Therefore, I implemented joint limits on the robot. This way, the robot cannot plan trajectories that are impossible in real life. When implementing, with the best case tuned PI control, it worked very well and realistically. I did not have to tune my PI control much with the joint limits added. However, after changing the PI gains, I also changed them to be the same in the best no joint limits case, just to see a true comparison of the addition of joint limits.

For all of my joint limit cases, I chose to implement joint upper and lower bounds on joints 1, 2, 3, and 4. I chose all the joints but joint 5 because we command the end-effector to be at a specific angle during the grasping motion, so it didn't seem as prevalent to whether the robot would have any self-collisions or singularities. As for the other four joint limits, as mentioned above I used the sliders from Scene 3 in CoppeliaSim to see what joint angles got the robot in a position close to its desired trajectory positions and then bounded the joints based on if I thought it could self-collide and/or get to a singularity. I knew that my robot arm never needed to extend straight up and/or flip over, so I bounded my limits to prevent the robot from doing that. From that the measure of realism of my robot's trajectory significantly improved. The addition of joint limits in each of my three cases demonstrate the importance of realistic modeling. Even though it can be more difficult, there is only so far you can get with an unrealistic model in terms of usefulness.

As for my overshoot case, it was very difficult to find PI gains that lead the robot to overshoot while still picking up the block and completing the task without steady-state error. I had to choose small deviations in the PI gains from the best case in order for this to work. If the proportional or integral gain terms were too big, the robot would lock into too many joint limits, and since the Jacobian of the respective column would be all zeros, many of the joints wouldn't be commanded and thus the robot couldn't pick up the block. I also had to change the initial end-effector transformation height, z term of T_se_init, to be larger so that the joints wouldn't lock at the limits as quickly. The overshoot occurred for the chassis body and can be seen in the graph, however it is hard to be seen in the simulation. When there were no joint limits, the overshoot is much easier to see because I could command much larger PI gain terms and the arms would spin around itself and oscillate back to the desired position and orientation in order to pick up the block.

As for the new task case, I left the PI gain terms the same as the best case PI terms, so that I could see if my best PI control could handle a new case. All I changed was the start and end positions of the block, as well as increased the height of initial end-effector transformation height, z term in T_se_init, like in the overshoot case. I changed the initial block position to be x=-0.5, y=1.0, and I changed the final block position to be x=1.0, y=-0.5. For the no joint limits case, the robot still chose a trajectory where the arm went through itself in order to grab the block. However, when I had the joint limits implemented, the robot found a realistic trajectory to perform while still grabbing the block and placing it in the correct position.

In more detail, an important and unexpected problem I encountered was while testing the overshoot case with joint limits. I tried to get the robot to overshoot a lot, by increasing the proportional gain, but making the proportional gain too high caused the robot to shoot to its joint limits, and never update since the Jacobian is always zero, and ultimately the robot failed to pick up the block. I didn't even think about this happening until I saw it in simulation, where I realized the restriction on joint limits was causing this failure. I therefore had to make the proportional gain still relatively small in order to overshoot but still pick up the block for my joint limits. I also had to change my initial transform of the end-effector with respect to the world for the trajectory generation. I increased the z position from 0.25 to 0.5. I had to do this because the lower z height would cause the arm to hit its joint limits before reaching the block and not be able to send commands to many joints in the arm to have it grab the block. Therefore, in my overshoot graphs, there is very little overshoot.