MAE 204 - Robotics - Final Project

Prof. Michael Tolley

Due: Thursday March 21, 2024 at 11:59pm

Logistical notes on the final project

- This project will count as the final exam for MAE 204 (worth 35% of your course grade)
- You are welcome (but not required) to find a partner to work with in writing the code required for the final project. Working with another student may help you detect and correct bugs in your code. Students are responsible for coordinating with their partners and paired partners will have a single code submission. Regardless of if you write the code with a partner, each student should submit their own final report.
- Your final submission should be a reflection of your own effort (working with your partner on coding if you have one). You are welcome to meet and discuss this project and any roadblocks you are facing with your classmates, friends, the course staff, etc. You are also welcome to use the "Modern Robotics: Mechanics, Planning, and Control Code Library" that accompanies the textbook. Your are not, however, allowed to directly copy code from other students or elsewhere. This would defeat the purpose of taking the class!
- Any submitted code may be compared using the Method of Software Similarity test developed at Standford (https://theory.stanford.edu/ aiken/moss/) and instances of excessive similarity will be investigated and may be referred to the UC San Diego Academic Integrity Office (https://academicintegrity.ucsd.edu/).
- Any final results (including videos, plots, descriptions, etc.) must be generated from your submitted code, unless the discrepancy is clearly justified in your final report (see "Final Submission" below). Other discrepancies may be referred to the UC San Diego Academic Integrity Office.
- The project will take roughly 20 hours to complete (depending on your team's comfort with coding). The first two Milestones must be submitted by the dates listed below. These Milestone submissions will be graded for completeness (i.e., you will receive full credit just for submitting the required components). Thus, you will receive 10% of your final project grade just for submitting a reasonable attempt to each Milestone by the dates noted below! You are allowed to change the code submitted for these milestones for your final submission if you notice any bugs or other problems with them.

Project Details

Summary In this project, you will write the software required to control the motion of a mobile manipulator robot called a youBot composed of a mobile base with four Mecanum wheels and a 5R arm. This software will consist of four main components (described in more detail below):

1. A Kinematics Simulator (Tested in *Milestone 1*, and included in final submission)
2. A Reference Trajectory Generator (Tested in *Milestone 2*, and included in final submission)
3. A Feedforward plus Feedback Controller (included in final submission)
4. A wrapper script that calls the above three components (included in final submission)

The final output of your software will be a comma-separated values (csv) text file that specifies the configurations of the chassis and the arm, the angles of the four wheels, and the state of the gripper (open or closed) as a function of time. This specification of the position-controlled youBot will then be "played" on the CoppeliaSim simulator to see if your trajectory succeeds in solving the task.

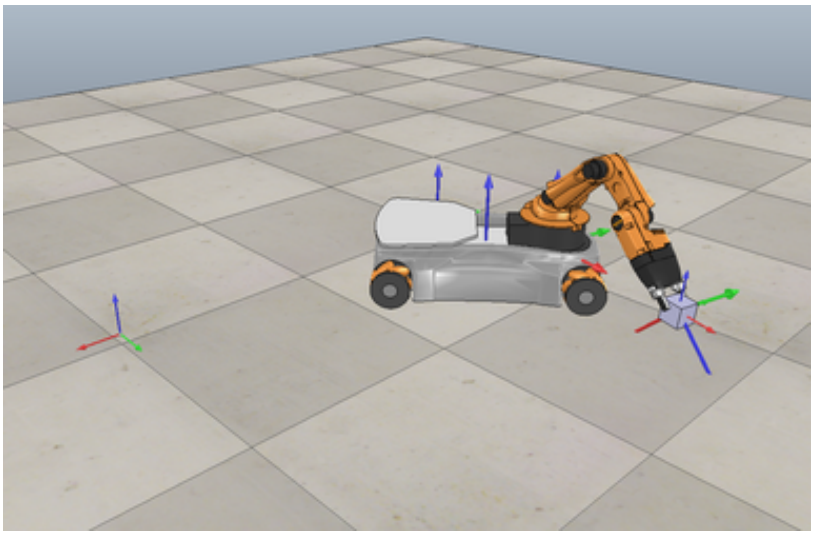

Figure 1: Image of youBot picking up the cube at the initial location, to move it to the target location (the frame seen on the left).

The Task You will test your control code on the mobile manipulation task of moving to an object (a cube), picking it up, and placing it in a target position.

Approach In writing your code for this project, you will implement most of the main concepts learned in 204. To do this, you are free to use any of the functions in the code library accompanying the Modern Robotics textbook. Instead of testing your code to control a physical robot (which is challenging in a one-quarter course due to logistical considerations), you will test your code using CoppeliaSim to simulate the motion of the robot. To do this, you will use the demonstration Scene 6 (you can download and find instructions on the same page that you downloaded the scenes for HW6: http://hades.mech.northwestern.edu/index.php/CoppeliaSim_Introduction). Note that the only physical interactions that this scene will simulate are between the floor, the gripper, and the block it is trying to pick up (e.g., it does not check for joint limits, self-collisions, for interactions between the block and the wheels or body of the robot, etc.). Also, the only bodies that respond dynamically in this scene are the gripper fingers and the block. Everything else will follow the trajectories determined by the `.csv` file that you load. In addition to Scene 6, you will also use the MR Scene 8 to test your trajectory generator (Milestone 1).

Additional Resources The technical details for the final project for this course are similar to the Mobile Manipulation Capstone from the MR textbook author Prof. Kevin Lynch's course on Robotic Manipulation. The above link has many more details on many aspects of the project which may provide a useful reference if you get stuck. The textbook author has also made this video summary of the project which may be helpful. However, for project submission details for MAE 204, please consult this document.

Component 1: Kinematics Simulator (`NextState`)

Related textbook sections: 13.2, 13.4

For this component, you will write a function called `NextState` that uses the kinematics of the youBot (see MR Exercise 13.33), your knowledge of velocity kinematics, and your knowledge of the Euler method to predict how the robot will move in a small timestep given its current configuration and velocity. Thus, your function `NextState` should take as inputs:

Inputs

- The current state of the robot (12 variables: 3 for chassis, 5 for arm, 4 for wheel angles)
- The joint and wheel velocities (9 variables: 5 for arm $\dot{\theta}$, 4 for wheels u)
- The timestep size Δt (1 parameter)
- The maximum joint and wheel velocity magnitude (1 parameter)

Outputs `NextState` should also produce the following outputs that describe the configuration of the robot one timestep (Δt) later:

- The next state (configuration) of the robot (12 variables)

Approach The function NextState is based on a simple first-order Euler step:

- new arm joint angles = (old arm joint angles) + (joint speeds)Δt
- new wheel angles = (old wheel angles) + (wheel speeds)Δt
- new chassis configuration is obtained from odometry, as described in Chapter 13.4

Testing your `NextState` function To test your `NextState` function, you should embed it in a program that takes an initial configuration of the youBot and simulates constant controls for one second. For example, you can set Δt to 0.01 seconds and run a loop that calls `NextState` 100 times with constant controls $(u, \dot{\theta})$. Your program should write a `.csv` file, where each line has 13 values separated by commas (the 12-vector consisting of 3 chassis configuration variables, the 5 arm joint angles, and the 4 wheel angles, plus a "0" for "gripper open") representing the robot's configuration after each integration step. Then you should load the `.csv` file into the *CSV Mobile Manipulation youBot CoppeliaSim* scene (Scene 6) and watch the animation of the constant controls to see if your NextState function is working properly (and to check your ability to produce a `.csv` file).

Component 2: Reference Trajectory Generator (`TrajectoryGenerator`)

Related textbook sections: 9.1, 9.2

For this component, you will write a function called `TrajectoryGenerator` to create the reference (desired) trajectory for the end-effector frame $\{e\}$. This trajectory should consist of eight concatenated trajectory segments, described below. Each trajectory segment begins and ends at rest. This function is likely to use either `ScrewTrajectory` or `CartesianTrajectory` from the Modern Robotics code library. The following are suggested inputs and outputs for `TrajectoryGenerator`:

Inputs

- The initial configuration of the end-effector: $T_{se,initial}$
- The initial configuration of the cube: $T_{sc,initial}$
- The desired final configuration of the cube: $T_{sc,final}$
- The configuration of the end-effector relative to the cube while grasping: $T_{ce,grasp}$
- The standoff configuration of the end-effector above the cube, before and after grasping, relative to the cube: $T_{ce,standoff}$
- The number of trajectory reference configurations per 0.01 seconds: k. The value k is an integer with a value of 1 or greater. [1]

Outputs

- A representation of the N configurations of the end-effector along the entire concatenated eight-segment reference trajectory. Each of these N reference points represents a transformation matrix T_{se} of the end-effector frame $\{e\}$ relative to $\{s\}$ at an instant in time, plus the gripper state (0 for open or 1 for closed). These reference configurations will be used by your controller. Note: if your trajectory takes t seconds, your function should generate $N = t \cdot k/0.01$ configurations.
- A `.csv` file with the entire eight-segment reference trajectory. Each line of the `.csv` file corresponds to one configuration T_{se} of the end-effector, expressed as 13 variables separated by commas. The 13 variables are, in order:

  $r_{11}, r_{12}, r_{13}, r_{21}, r_{22}, r_{23}, r_{31}, r_{32}, r_{33}, p_x, p_y, p_z$, $gripper\_state$

[1] Although your final animation will be based on snapshots separated by 0.01 seconds in time, the points of your reference trajectory (and your controller servo cycle) can be at a higher frequency. For example, if you want your controller to operate at 1000 Hz, you should choose k = 10 (10 reference configurations, and 10 feedback servo cycles, per 0.01 seconds). It is fine to choose k = 1 if you'd like to keep things simple.

Trajectories

1. Move the gripper from its initial configuration to a "standoff" configuration a few cm above the block.
2. Move the gripper down to the grasp position.
3. Close the gripper[2].
4. Move the gripper back up to the "standoff" configuration.
5. Move the gripper to a "standoff" configuration above the final configuration.
6. Move the gripper to the final configuration of the object.
7. Open the gripper.
8. Move the gripper back to the "standoff" configuration.

Block details The block is a 5 cm × 5 cm × 5 cm cube, and it's frame, $\{c\}$ is at the center of the cube with axes aligned with the edges of the cube. The initial configuration of the block is at $(x, y, \theta) = (1 \text{ m}, 0 \text{ m}, 0 \text{ rad})$ and the final configuration is $(x, y, \theta) = (0 \text{ m}, -1 \text{ m}, -\pi/2 \text{ rad})$

Testing your `TrajectoryGenerator` function You should use the CoppeliaSim scene (Scene 8) to help you test your `TrajectoryGenerator` function. This scene reads in your `.csv` file and animates it, showing how the end-effector frame moves as a function of time. You should verify that your `TrajectoryGenerator` works as you expect before moving on with the project. This video shows an example of a typical output of your trajectory generator function, as animated by CoppeliaSim Scene 8.

Component 3: Feedforward Plus Feedback Control (`FeedbackControl`)

Related texbook sections: 4.1.2, 5.1.2, 11.3, 13.5

The third component you will write is a function, `FeedbackControl` that calculates the task-space feedforward plus feedback control law discussed in class, and also appearing as Equation (11.16) and (13.37) in the textbook. The following are suggested inputs and outputs:

Inputs

- The current *actual* end-effector configuration X (aka T_{se})
- The current *reference* end-effector configuration X_d (aka $T_{se,d}$)
- The *reference* end-effector configuration at the next timestep, $X_{d,next}$ (aka $T_{se,d,next}$)
- The PI gain matrices K_p and K_i
- The timestep Δt between reference trajectory configurations

Outputs

- The commanded end-effector twist $\mathcal{V}$ expressed in the end-effector frame $\{e\}$ (for plotting purposes).
- The commanded wheel speeds, u and the commanded arm joint speeds, $\dot{\theta}$.

[2]Note: the closing of the gripper is governed by the dynamics of the scene simulation. All you do on the trajectory is include a "1" as the gripper state to indicate that you want the gripper to close, but this will take up to 0.625 s. Thus, you will need at least 63 consecutive lines of the same $\{e\}$ configuration with a "1" in the gripper state to ensure that the gripper closes completely. The same is true for opening.

Control Law To calculate the control law, `FeedbackControl` needs the current actual end-effector configuration $X(q, \theta)$, which is a function of the chassis configuration q and the arm configuration θ. The values (q, θ) come directly from the simulation results (Component 1). In other words, we're assuming perfect sensors.

The error twist X_{err} that takes X to X_d in unit time is extracted from the 4×4 se(3) matrix $[X_{err}] = \log(X^{-1}X_d)$. `FeedbackControl` also needs to maintain an estimate of the integral of the error, e.g., by adding $X_{err}\Delta t$ to a running total at each timestep. The feedforward reference twist $\mathcal{V}_d$ that takes X_d to $X_{d,next}$ in time Δt is extracted from $[\mathcal{V}_d] = (1/\Delta t)\log(X_d^{-1}X_{d,\text{next}})$.

The feedback control law gives the commanded end-effector twist $\mathcal{V}$ expressed in the end-effector frame $\{e\}$. To turn this into commanded wheel and arm joint speeds $(u, \dot{\theta})$, we use the pseudoinverse of the mobile manipulator Jacobian $J_e(\theta)$:

$$\begin{bmatrix} u \\ \dot{\theta} \end{bmatrix} = J_e^{\dagger}(\theta)\mathcal{V}$$

Testing your `FeedbackControl` function Before moving on, it is strongly recommended to test that your `FeedbackControl` function is working properly. While this step is not required for your submission, it may save you a lot of time debugging later.

To confirm that you are correctly calculating $\mathcal{V}$ and the controls $(u, \dot{\theta})$ using the Jacobian pseudoinverse. Here are some test inputs you should try (see Fig. 2):

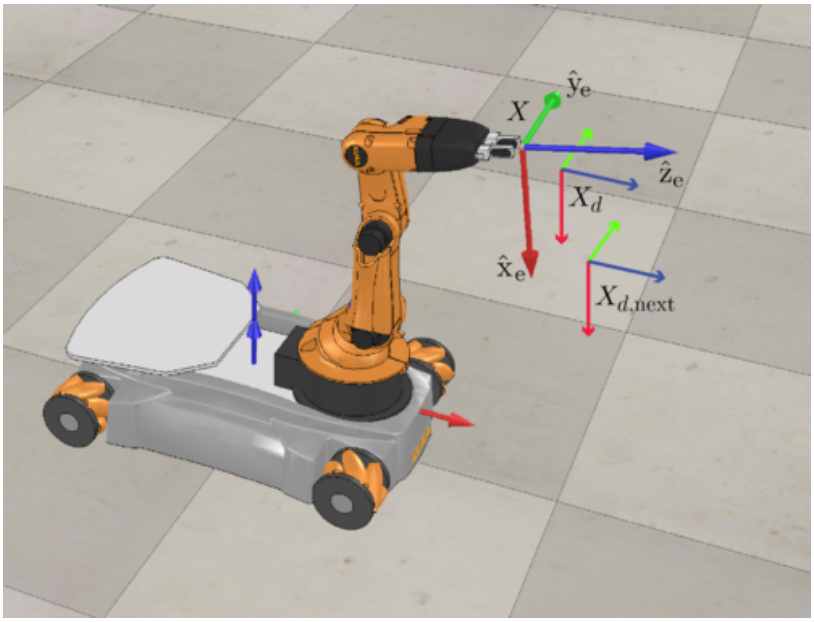

Figure 2: This figure illustrates a recommended test for your `FeedbackControl` function. The current robot configuration (q, θ) is shown, as well as the current end-effector configuration (X), the current reference end-effector configuration (X_d), and the reference end-effector configuration a time Δt later $(X_{d,next})$.

- robot configuration: $(\phi, x, y, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = (0, 0, 0, 0, 0, 0.2, -1.6, 0)$. In other words, all configuration variables are zero except arm joint angle 3, which is 0.2 radians, and arm joint angle 4, which is -1.6 radians.
- $X_d = \begin{bmatrix} 0 & 0 & 1 & 0.5 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- $X_{d,\text{next}} = \begin{bmatrix} 0 & 0 & 1 & 0.6 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- $X = \begin{bmatrix} 0.170 & 0 & 0.985 & 0.387 \\ 0 & 1 & 0 & 0 \\ -0.985 & 0 & 0.170 & 0.570 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ (this is calculated from the robot configuration given above)
- K_p and K_i matrices are zero matrices
- $\Delta t = 0.01$

With these inputs, your program should give you

- $\mathcal{V}_d = (0, 0, 0, 20, 0, 10)$

  Note: From Fig. 2, it should make sense that only the linear components are nonzero, since the orientations of X_d and $X_{d,next}$ are the same. These linear components are in the x and z directions of the X_d frame, and the x component is larger.

- $[\mathrm{Ad}_{X^{-1}X_d}]\mathcal{V}_d = (0, 0, 0, 21.409, 0, 6.455)$

  Note: this step expresses the above twist in the current frame of the end-effector (X). It should be clear from Fig. 2 why there are still only x and z components, but the x component has become a bit larger, and the z component a bit smaller.

- $\mathcal{V} = (0, 0, 0, 21.409, 0, 6.455)$

  Since the feedback gains are set to zero, the commanded twist is identicle to $[\mathrm{Ad}_{X^{-1}X_d}]\mathcal{V}_d$.

- $X_{\mathrm{err}} = (0, 0.171, 0, 0.080, 0, 0.107)$

  To drive from X to X_d, we must rotate in the positive $\hat{y}_e$ direction, and move linearly in the positive $\hat{x}_e$ and $\hat{z}_e$

- The increment to the numerical integral of the error should be $X_{err}\Delta t$

- $J_e = \begin{bmatrix} 0.030 & -0.030 & -0.030 & 0.030 & -0.985 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 \\ -0.005 & 0.005 & 0.005 & -0.005 & 0.170 & 0 & 0 & 0 & 1 \\ 0.002 & 0.002 & 0.002 & 0.002 & 0 & -0.240 & -0.214 & -0.218 & 0 \\ -0.024 & 0.024 & 0 & 0 & 0.221 & 0 & 0 & 0 & 0 \\ 0.012 & 0.012 & 0.012 & 0.012 & 0 & -0.288 & -0.135 & 0 & 0 \end{bmatrix}$

- $(u, \dot{\theta}) = (157.2, 157.2, 157.2, 157.2, 0, -652.9, 1398.6, -745.7, 0)$

  The calculated controls drive the wheels forward at equal speed (to move the chassis forward), and only arm joints 2, 3, and 4 rotate, since joints 1 and 5 do not contribute to the required $\mathcal{V}$. *Note: the joint speeds in this illustrative example are unreasonably large!*

If your K_p matrix is the identity matrix instead of zero, then you should get $\mathcal{V} = (0, 0.171, 0, 21.488, 0, 6.562)$ and $(u, \dot{\theta}) = (157.5, 157.5, 157.5, 157.5, 0, -654.3, 1400.9, -746.8, 0)$.

If you don't get these results, you should correct your program before moving on!

Avoiding singularities, and implementing joint limits to avoid self-collisions Two optional but recommended improvements for your `FeedbackControl` function are:

- Specify a tolerance (optional) when you call a psuedoinverse function (e.g., `pinv()` in Matlab) to treat small singular values as zero, avoiding unreasonably large entries in the Jacobian psuedoinverse.
- Implementing joint limits to avoid self-collisions and singularities (e.g. when Joints 3 or 4 are at an angle of zero).

If you need suggestions on how to implement these improvements, you can see the sections *Singularities*, and *Implementing joint limits to avoid self-collisions and singularities* at this link.

Component 4: Full Program/Wrapper Script

Finally, you will write the wrapper code that calls the previous three components to generate a reference trajectory, simulate robot motion, and compute control inputs to achieve the desired motion. The first step is to use `TrajectoryGenerator` to generate the trajectory. Your program should then loop through the steps of the N reference configurations, using them as X_d and $X_{d,next}$ to calculate the feedforward twist $\mathcal{V}$ (note you will only be able to do this $N-1$ times.) Each time through the loop, the program should:

- calculate the control law using `FeedbackControl`, and generate the wheel and joint controls using $J_e^\dagger(\theta)$.
- use `NextState` to calculate the new configuration
- store every k^{th} configuration for later animation
- store every k^{th} X_{err} 6-vector to later plot the evolution of the error over time

Once the program has completed all iterations of the main loop, it should write the `.csv` file of configurations (e.g. the file should have 1500 lines for a total motion of 15 s). You should then load the file into the CoppeliaSim Scene 6 to visualize the results, and plot your X_{err} data.

Inputs/Initializations

- the initial resting configuration of the cube frame
- the desired final resting configuration of the cube frame
- the actual initial configuration of the youBot
- the reference initial configuration of the youBot (different from actual to test feedback control)
- gains for your feedback controller

Outputs

- a `.csv` file that can be played through the CoppeliaSim scene. Each line of the file should have the following 13 entries where the J's are joint angles and the W's are wheel angles:

  `chassis phi, chassis x, chassis y, J1, J2, J3, J4, J5, W1, W2, W3, W4, gripper state`

- a data file containing the 6-vector end-effector error as a function of time

Testing feedforward control Once you have your full program assembled, you should make sure that feedforward control works as expected before testing feedback control. Choose an initial configuration of the robot that puts the end-effector exactly at the configuration at the beginning of the reference trajectory. Run your program with $K_p = K_i = 0$ (i.e., feedforward control only). This should result in a `.csv` file which, when played through the mobile manipulation capstone scene, drives the robot to pick up the block and put it down at the desired configuration. (Or at least it should come close to doing so! Small numerical errors in the integration will be fixed when you add a feedback controller.)

If your robot does not follow the desired trajector with feedforward control only, it is time to start debugging! Your end-effector reference trajectory must be correct, if you already tested Milestone 2. So now you have to figure out why the wheel and arm controls that your feedforward controller and Jacobian pseudoinverse are generating do not drive the end-effector along the reference trajectory. You could try removing joint speed limits if these are preventing the robot from following the planned trajectory, or increase the time of the trajectory so large joint speeds are not needed to follow the trajectory.

You should also try starting the end-effector with some initial error from the reference trajectory, but still only use feedforward control. See how the end-effector moves under these circumstances. Does it make sense to you?

Do not move on with the project until your feedforward control works as you expect. Otherwise the effects of PI feedback control will only further confuse the situation.

Testing feedforward plus feedback control To test the full feedfoward plus feedback control, use the default initial and goal configurations for the cube in the capstone CoppeliaSim scene, i.e., $(x, y, \theta) = (1\text{ m}, 0\text{ m}, 0\text{ rad})$ and $(x, y, \theta) = (0\text{ m}, -1\text{ m}, -\pi/2\text{ rad})$. Let the initial configuration of the end-effector reference trajectory be at

$$T_{se} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Choose an initial configuration of the youBot so that the end-effector has at least 30 degrees of orientation error and 0.2 m of position error. Try executing the feedforward controller ($K_p = K_i = 0$) and play the resulting `.csv` file through the CoppeliaSim capstone scene to see what happens.

Now add a positive-definite diagonal proportional gain matrix K_p while keeping the integral gains zero. You should keep the gains "small" initially so the behavior is not much different from the case of feedforward control only. As you increase the gains, can you see some corrective effect due to the proportional control?

Eventually you will have to design a controller so that essentially all initial error is driven to zero by the end of the first trajectory segment; otherwise, your grasp operation may fail.

Once you get good behavior with feedforward-plus-P control, try experimenting with other variants: P control only; PI control only; and feedforward-plus-PI control.

Deliverables

Milestone 1 [5 points] (Due 11:59pm, Tuesday, March 12, 2024)

You should test your `TrajectoryGenerator` as described above above under *Component 2*. You will submit a .pdf file containing your `TrajectoryGenerator` code, along with a link to a video capture of a CoppeliaSim simulation demonstrating successful completion of the task (as you did for HW5) through Gradescope. This will be graded for completeness, not for correctness (i.e., a grade of 5/5 does not imply that your code is perfect). You are allowed to update your code for your final submission if you find any problems/errors. This milestone may be submitted with a partner in 204.

Milestone 2 [5 points] (Due 11:59pm, Tuesday, March 14, 2024)

You should test your `NextState` function as described above above under *Component 1*. You will submit a .pdf containing the code described above (your `NextState` function plus wrapper code) along with a link to a video capture of a CoppeliaSim simulation demonstrating that your function is working properly for constant control inputs. This will be graded for completeness, not for correctness (i.e., a grade of 5/5 does not imply that your code is perfect). You are allowed to update your code for your final submission if you find any problems/errors. This milestone may be submitted with a partner in 204.

Final Submission [90 points] (Due 11:59pm, Thursday, March 21, 2024)

Your final submission will be submitted through Gradescope, and should include an individual `.pdf` submission, and a code submission (may be submitted with a partner) containing your code required to complete Component 3 and Component 4 (including the `NextState` and `TrajectoryGenerator` functions).

Report Summary Your main project file will be an individual `.pdf` report. The first section of your report should briefly explain your software and your approach. If you needed to follow a different approach to solve the problem than the one described above, explain why and explain your solution method. If you encountered anything surprising, or if there is something you still don't understand, or if you think an important point is neglected in the description of the project, explain it. If you implemented singularity avoidance, joint limit avoidance, or any other enhancement over the basic project description given on this page, explain your method. You may also wish to include more details in the results sections described below, showing the results when using your enhancements vs. when you don't use your enhancements, to highlight the value of your enhancements.

Results Your `.pdf` report should also include three sections describing the results of your code for three cases: "best", "overshoot", and "new task". The "best" and "overshoot" cases solve a pick-and-place task where the initial and final configurations of the cube are at the default locations in the capstone CoppeliaSim scene. The "new task"

should have different initial and final block configurations which you are free to choose. In all cases, the initial configuration of the end-effector should have at least 30 degrees of orientation error and 0.2 m of position error from the first configuration on the reference trajectory. The "best" case should describe results using a well-tuned controller, either feedforward-plus-P or feedforward-plus-PI. The convergence exhibited by the controller does not necessarily have to be fast (in fact, it is more interesting if the convergence is not too fast, so the transient response is clearly visible), but the motion should be smooth, with no overshoot, and very little error by partway through trajectory segment 1. The case "overshoot" should contain the results using a less-well-tuned controller, one that exhibits overshoot and a bit of oscillation. Nonetheless, the error should be eliminated before the end of trajectory segment 1. Your controller for "overshoot" will likely be feedforward-plus-PI or just PI. You can use any controller to solve the "new task".

For each of the three cases ("best", "overshoot", and "new task"), be sure to include:

1. A very brief description of the type of controller, the feedback gains, and any other useful information about the results. For the "new task", indicate the initial and goal configurations for the cube.

2. A link to a video of your `.csv` file being animated by the CoppeliaSim scene. (Please make the video viewable by anyone with the link! We will not share the link).

3. A plot of the six elements of X_{err} as a function of time, showing the convergence to zero.

Discussion In your report, please also answer the following questions (each answer should be no more than 1 paragraph):

1. What are the advantages and disadvantages of incorporating an integrator term to your controller (i.e., using Feedforward + PI control rather than Feedfroward + P control)? Why do we start observing overshoot once we add integrator to our controller?

2. If you reduce the maximum joint velocities to low values, you should see error increase again after picking up the cube. Why does this happen?

3. In this project, our controller directly prescribes the $\dot{\theta}$ to the joints of the robot. In what cases may this be possible on a real robot?

4. If we were to implement torque control for the youBot (rather than speed control), what additional information (i.e., input parameters and/or pre-defined constants) would the `FeedbackControl` function need to know to compute the required torques? Which function from the MR textbook code would it use?

Commented Code In addition to your .pdf submission containing your report, you will submit the code that your wrote to solve this project to a separate "code submission" section on Gradescope (may be with a partner). Your code should be commented, so it is clear to the reader what the code is doing. No need to go overboard with too many comments, but keep in mind that the person reading your code may not be fluent in your programming language. Your code comments must include an example of how to use the code, and to make the code easy to run, each separate task you solve should have its own script, so by invoking the script, the code runs with all the appropriate inputs. (This makes it easy for others to test your code and modify it to run with other inputs.) Apart from the scripts, only turn in functions that you wrote or modified; you don't need to turn in other MR functions that your code uses.

Project grading Your project will be graded on the clarity and correctness of your submission files and your code. Your results will be graded on their correctness, including the quality of your videos and whether your error plots show reasonable convergence to zero.

If you succeed in this project, congratulations! You have integrated concepts from throughout 204 in a fairly sophisticated piece of software!