Trajectory Tracking with Baxter

EECS C106B Project 1B

Osher Lerner Will Panitch Enya Xing

I. Methods

We demonstrate trajectory tracking with Baxter with the construction of three different controllers.

A. Workspace Velocity Controller Theory

Here, we receive the target trajectory we want the end effector to track in workspace configuration $g_d(t) \in SE(3)$. The main problem here is that Baxter is only able to take in jointspace commands. We strategize and first compute a body-frame workspace velocity $U^b(t)$ that we reduce the error on. From the target position and target velocity, we calculate g_d, our desired configuration. We then obtain Baxter's current position and calculate g_t, our current configuration and use the inverse of g_t to calculate g_{td}, the desired configuration in the frame of the current. We then calculate their respective adjoint matrices Ad_{g_t} and $Ad_{g_{td}}$. ξ_{td} is our velocity that reduces error between the target and desired configurations and is calculated through

$$\hat{\xi}_{td} = \log(g_{td})$$

We define K_p to be a 6x6 matrix of positive diagonal tuned controller gains and multiply it to ξ_{td} as our feedback and add it to our target velocity, now in the body frame. With our calculated ξ_{td}, and our target velocity in the desired frame as V_d^b, we assemble our Cartesian control law to be

$$U^b(t) = K_p\xi_{td} + Ad_{g_{td}}V_d^b$$

Our last step converts our control input into joint velocities that can be taken in by Baxter. We make this transformation into the spatial coordinates

$$U^s(t) = Ad_{g_t}U_b(t)$$

and conclude by multiplying U_s with our spatial Jacobian pseudo-inverse to obtain the joint velocities

$$\dot{\theta}(t) = J^\dagger(\theta)U^s(t)$$

After properly computing the frame transformations using custom mathematical operations, tuning this controller only consisted of increasing the gain until the desired trajectory was closely tracked. This controller tuned very easily, with a diagonal K_p of 1.0 for each spatial and rotational dimension generating a smooth circle as desired.

B. Jointspace Velocity Controller Theory

From our calculated trajectories, we are provided with a target position, target velocity, and target acceleration $(\theta_d(t), \dot{\theta}_d(t), \ddot{\theta}_d(t))$, all in joint space coordinates. We are also able to obtain a vector of Baxter's seven joints $\theta(t) \in \mathbb{R}^7$, and we decide on a control input in terms of joint velocities. The Baxter then tracks these velocities using an internal control loo.

We define our control input to be the equation

$$u(t) = u_{ff}(t) + u_{fb}(t)$$

where u_{ff} is our feedforward term and u_{fb} is our feedback term, with the following definitions

$$u_{ff}(t) = \dot{\theta}_d(t)$$
$$u_{fb}(t) = K_p e(t) + K_v \dot{e}(t)$$

where our error $e(t)$ in joint positions is defined as

$$e(t) = \theta_d(t) - \theta(t)$$

and K_p and K_d are the proportional and derivative terms for the PD controllers, respectively. $\theta_d(t)$ and $\theta(t)$ are our desired and actual joint positions.

The open-loop controller tells the Baxter the desired velocity for each joint at each timestep, and the added feedback term corrects for errors in tracking this velocity by adding a corrective velocity proportional to the error, as well as a term proportional to the derivative of the error to dampen oscillations.

We tuned the PD controller by increasing K_p from 0 until the robot oscillated about the desired trajectory. Then, we increased K_v until these oscillations were largely dampened. This controller also tuned pretty smoothly, resulting in the gains K_p of 1.0 along each joint dimension independently and K_v of 0.1 for each of the 7 joints.

C. Jointspace Torque Controller

The jointspace torque controller uses the given target position, velocity, and acceleration of the joints at any timestep to compute a torque input for each joint to track the desired trajectory. To convert between joint trajectory to input torque we utilize the differential equation derived from the open-chain manipulator's Lagrangian dynamics.

$$M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + N(\theta, \dot{\theta}) = \tau$$

Then — using $e(t) = \theta_d(t) - \theta(t)$ — by selecting our inputs to be

$$u = M(\theta)\left(\ddot{\theta}_d + K_v\dot{e} + K_p e\right) + C(\theta,\dot{\theta})\dot{\theta} + N(\theta,\dot{\theta})$$

we cause an effective differential equation over the error between the desired and current trajectories:

$$M(\theta)\left(\ddot{e} + K_v\dot{e} + K_p e\right) = 0$$

Since the inertia matrix is positive semi-definite and our constants are positive, this naturally drives the error to 0 and thus tracks the desired trajectories.

In reality, we do not know the true dynamics of the system. We used a calculation of the dynamics using the open-chain model and a gravity of $9.81 m/s^2$, but did not get a Coriolis term. Approximating the Coriolis term as 0 is reasonable so long as velocities are small (since it is multiplied by $\dot{\theta}$), so we generated slow desired trajectories (10s for line and circle, 20s for polygon) to remain within this regime. Furthermore, our computed gravity vector was too an order of magnitude too large and did not account for empirical lateral drift, so we manually tuned it to correct for observed drifts in each of the joints. We approximated these corrections as constant with respect to the joint state, which was accurate enough for our trajectories, especially when used in conjunction with a feedback term.

We tuned the PD controllers by increasing K_p from 0 until the robot oscillated about the desired trajectory. Then, we increased K_v until these oscillations were dampened. We found the trajectory was lagging behind at multiple points along the trajectory, so we increase K_p again, then tuned K_v appropriately. Raising K_p first allowed us to verify we were able to track the desired trajectory quickly enough before removing oscillations. We repeated this process several times until we had nearly exact tracking with some joints. For the other joints, we accounted for constant errors by correcting the G vector, then tuned their gains individually (the first three joints, especially the shoulder, needed a stronger push). The tuned gains were

$$K_p = \text{diag}(45, 40, 50, 30, 30, 30, 30)$$
$$K_v = \text{diag}(3, 3, 3, 3, 3, 3, 3)$$

II. Experimental Results

A. Experimental Design

We ran our experiments on a Rethink Robotics Baxter industrial robot. This robot has a seven degree-of-freedom arm with an interchangeable end-effector. This hardware was kept standardized between runs and trajectories, which helped eliminate any discrepancies that would be caused by differing hardware systems. In order to effectively test the differences between our controllers, we drove our manipulator arm using a number of distinct controller paradigms, each of which we created and tuned by hand (implementations available in appendix). These included **(A)**, a purely feed-forward controller, **(B)**, a tuned jointspace-velocity controller, **(C)**, a tuned workspace-velocity controller, **(D)**, a jointspace torque controller, and finally, **(E)**, a controller provided by the MoveIt! library, a professional control library, with parameters provided by experts. With each controller, we ran a set of three trajectories, which included **(1)**, a linear motion between two points in space, **(2)**, a circular motion around a predefined central point, and **(3)**, a polygonal trajectory consisting of a linked set of linear trajectories. Each of these trajectories was then observed, documented, and plotted to measure its deviation from the planned path, allowing us to qualitatively and quantitatively observe the differences between each controller. In addition, the inclusion of both a feed-forward controller and a professionally tuned controller served to act as a control group and a comparison group, respectively, with which we could contrast our own controllers.

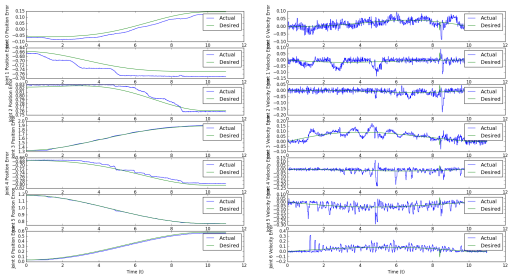

Fig. 1. position and real/intended velocity vs. Time for the torque controller on a linear trajectory

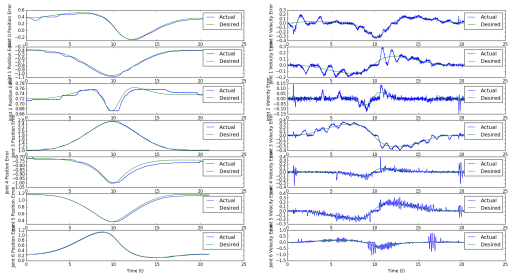

Fig. 2. position and real/intended velocity vs. Time for the torque controller on a circular trajectory

B. Experimental Result Plots

Each of our controllers was implemented first in a simple two-dimensional simulator before being implemented on the robot. This allowed us to check the theoretical viability of its planning capabilities before running it on our expensive and fragile real-world equipment. This approach helped us develop a conceptual understanding of the controllers and ensure the safety of ourselves, our peers, and our equipment. When moving onto the physical robot, we set a hard-coded limit on the torque output of our controller to avoid extreme anomalous movements while tuning.

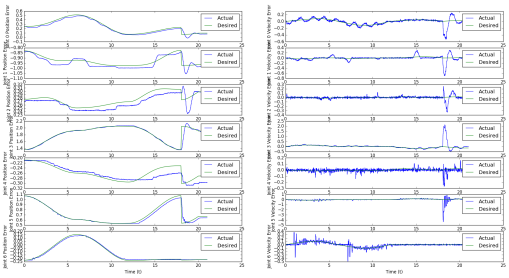

Fig. 3. position and real/intended velocity vs. Time for the torque controller on a polygonal trajectory

C. Comparison with the MoveIt! Controller

Each of the controllers that we created had a number of strengths and weaknesses, which we experienced firsthand as we designed, implemented, and ran each one of them. In addition, these differences were not entirely translatable from our simulation environment, which lead to a few unexpected challenges.

For example, our feed-forward controller, which performed relatively well in the simpler simulation environment, had much more trouble in the real world. This meant poor trajectory-tracking, as well as a few dangerous situations, such as times that the robot nearly collided with itself in attempting to reach the positions it believed were correct. Its motions were generally smooth, with solid performance on the line task, but this smoothness came at the cost of far poorer trajectory tracking, especially in the case of the polygonal path, which included repeated acceleration and deceleration, as well as sharper corners to turn.

The jointspace velocity and workspace velocity controllers were both relatively robust, both using a uniform K_p of 1 and a K_v of 0.1. The performance between the two was highly comparable, given their shared method of feedback, with perhaps a slight advantage to the jointspace controller. Indeed, from a programming perspective, this advantage was compounded further by the relative simplicity of the jointspace controller, which required far fewer frame conversions in order to work effectively. These two controllers seemed to be the best of the hand-implemented choices for our simple tasks, with performances that hewed most closely to that of the professionally designed controller.

Before tuning, the joint-torque controller was the least robust in its performance across the trajectories. In fact, unlike the other controllers that we implemented, the joint-torque controller resulted in paths which bore little-to-no resemblance with the intended ones unless feedback was included. Indeed, the feedback term came to dominate the tuned torque controller, which resulted in noisy paths which occasionally lagged behind the given motions. However, after the inclusion of a large K_p matrix (on the order of 40–50), and a smaller K_v term (on the order of 3–5), we were able to generate a controller that reliably reproduced the motions and traced the correct paths with the tip. However, we found that this controller, while able to accurately reproduce the intended paths, was still a little bit jerky and produced less uniform results than the other controllers.

MoveIt!, our professionally designed controller, was extremely robust in nearly all cases. It managed to output smooth and fast trajectories for all three of our test cases, and it was the most robust when taking into account safety concerns, as it has a built-in method for catching unreasonable control inputs. In comparison to the open-loop controller, it had far less difficulty in accurately tracking intended trajectories, and was far more capable of handling acceleration and deceleration motions. The jointspace and workspace velocity controllers were much better at accurate tracking, but even after tuning, neither was able to produce the same smooth results as MoveIt! at high or low velocities. These advantages of MoveIt! make it our recommendation for execution of any future trajectories, but it is not without its drawbacks. However, due to its more stringent system of constraints, it was unable to find kinematic solutions to reach a number of paths that all of our other controllers had no difficulty in executing.

III. Applications

Torque control can be best applied to pick and place processes or positioning of fragile objects with variation. By specifying specific torques, we have more refined and predictable control of the robot which is much needed to handle objects. The downsides of torque control lie in the fact that we need accurate models of the dynamics, and have to give up our position and velocity controls.

Joint-space velocity control is best for contexts in which the dynamics are too difficult to properly model. Large trajectories with changing masses, such as catching and throwing objects with the Baxter, would have varying effects on each joint impossible to model precisely. This direct controller will simply push the observed state of each joint to the desired one, whereas the computations of our other controllers could fail to find the right inputs due to the changing dynamics. This controller is also the fastest to compute, which can be critical for high frequency controllers on embedded computers. An example of such a context would be controlling drone propellers, which operate on rapid scales using minimal operations since the hardware is severely limited by weight and battery.

Workspace control is best for tasks that require precise interaction with the environment outside of the robot. These include applications such as docking a spaceship, inserting a key into a lock, or drawing. What is especially useful is the ability to separately tune controllers for different spatial directions and rotations. We can have a very tight controller on the orientation and the z-coordinate, while using a slower more dampened controller for the x and y spatial directions.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

IV. Appendix

Link to GitHub repository with implementations: https://github.com/ucb-ee106-classrooms/project-1-swag

Link to Google Drive with Videos: https://drive.google.com/drive/folders/11aKh98keORzV5QOgIGBNS_EhfG3zYZc6?usp=sharing

We enjoyed the learning goals of this assignment! We got to really understand the different types of controllers by implementing and tuning each, and it gave us a solid foundation to be tracking trajectories we generated ourselves. However, we did find that much of the process of tuning involved wrestling with ROS setup, strange bugs, and weird behaviors not relevant to the theory. We know that this is part of any real-world robotics research, but there were admittedly a number of frustrating moments in which we wished that there was an easier way to interact with the conceptual material. To be more specific, we encountered a number of unexpected behaviors in the implementations of Python and Numpy that the lab computers are equipped with. For example, a number of Numpy's dimension alteration functions did not perform as expected, which forced us to spend a large amount of time debugging when the behavior of classic functions such as ravel and reshape did not work as they conventionally would. We encountered a number of these issues over the course of the project, and they proved especially difficult to track down, due to the use of a deprecated Python version. Additionally, the gravity matrix that was provided in the starter code seemed to be significantly off, and caused some very unexpected behavior, that had to be compensated for through far stronger feedback mechanisms than we were expecting. Indeed, to make our final controller operable, we ended up divining our own gravity vector based on the behavior of the robot at rest, which detracted from our overall understanding of the computed torque controller. Overall, however, this was an extremely fun project, and we are grateful to course staff for their hard work and understanding during this crazy semester :)