MAE 204 Final Report

MAE 204

Cody Pappa (cpappa@ucsd.edu)

Overview / Summary of code

I had a ton of fun working on this project. I thought the challenge was right within grasp and we were set up well to complete it. I believe I followed the coding instructions as closely as possible and it worked out well. Having separate milestones with well defined inputs and outputs helped to break down this large task into quite manageable chunks.

My code starts with defining the input variables like cube starting and end positions, the pickup and standoff positions, and the idealized starting E-E position. The other important starting variables is the 1st Joint positions, which through FK gives the actual E-E position. This is what is adjusted to start the robot off its idealized path.

Other important constants include Tsb, Tb0, the Screw axes, and the M matrix. Some variables that can be changed are the dt, k, limits, and the gains for feedback control, Kp and Ki. One issue I encountered, when I had all the joint limits set to 10, during feedforward only control the robot would go off-course rather quickly and end up far away from intended position. If I added feedback control, the robot did better. After lots of testing, I figured out the joint limit of 10 was set too low, and the robot could not keep up with the commanded speeds to follow the feedforward path.

The next section of code was used during feedforward testing. It was used to find actual joint positions based on the commanded start E-E config. Using this piece and only feedforward control, the robot could preform the task well. This code is turned off because we want to add some starting error to see our feedback controller work.

The Odometry section was taken right out of the book to map changed in PHI, X,Y to changes in wheel velocities

Next, we use the code from component 2 TrajectoryGenerator to create ideal E-E paths to follow. The only change I made to the directed inputs/outputs was to add a input vector for the times to each trajectory. Simply pass in a vector that controls how long each action should take. This could be adjusted to increase time and therefore decrease joint velocities to avoid saturation of the velocities. This creates 1500 E-E positions in a row for my case, as the total time of the action was set to 15 seconds.

Finally, we set up the loop to follow the trajectories. I started each iteration by finding Tsb, where the base of the robot is in space. Then T0e, which is where the E-E is in relation to its base. From these two calculations, and Tb0 which related the base to the arm, the final Tse is the E-E location in the S frame. X = Tsb*Tb0*T0e is the config of the E-E currently from the robot's previous moves. Next, use component 3 to calculate the error twist, Ve, from the found X and your generated trajectories. This Ve is calculated from the feedforward twist, the proportional error term, and the integral error term, dependent on the values of Kp and Ki. Once error twist is found, use the Jacobian to learn how to move the joints to apply that twist. I implemented a lower tolerance on the Inverse Jacobian calculation that treats any values less

than .0001 as a zero, so that singularities are avoided. Using the pseudo inverse of the Jacobians and the error twist, the speeds of the joints are found. The current joint positions, speeds, and dt are loaded into component 1 code NextState to output the next joint angles. From these joint angles, a crude joint limit violations check occurs. If any of the defined joints cross past those joint limits, the Jacobian for that joint is zeroed out. This stops the robot from using those joints to reach the desired position. Once the limits checks are done, the speeds and positions need to be recalculated based on the limited Jacobian, and these can be recorded and passed on to the next iteration of the loop. Every kth config is exported for animation, but I just left k at 1 so that every frame was animated

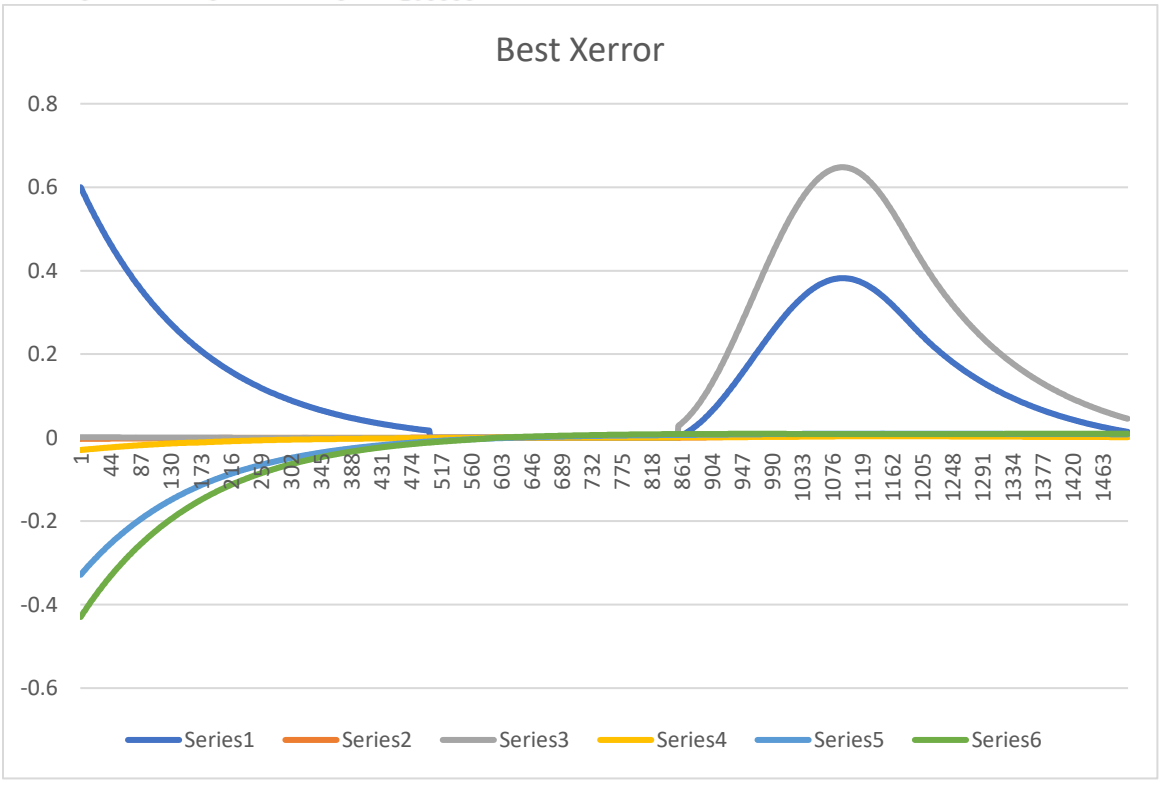
Results

Best

This test shows the feedforward + feedback controller working. See the desired starting position Xd vs the actual starting position X. With this test, the controller is starting with over .7 positional error and 30 degrees of orientation error. The Kp = I*0.6 and the Ki = I*.01. You can see the errors are quickly driven to 0 before grabbing the block, then they increase during the short move of positioning the block. These errors might be reduced with further tuning or simply by increasing the time of that move to allow the controller more time to drive down errors.

```
X =

   -0.0026   -0.5646    0.8253    0.3087
   -0.0018    0.8253    0.5646    0.4343
   -1.0000         0   -0.0032    0.4698
         0         0         0    1.0000
```

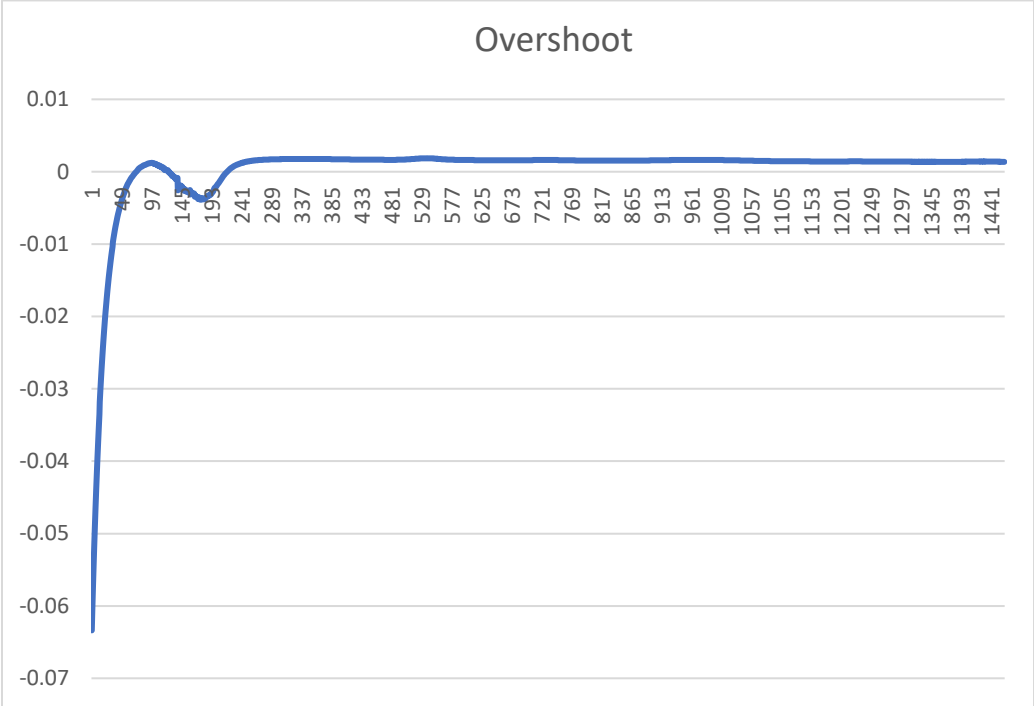
```
Xd =

         0         0    1.0000         0
         0    1.0000         0         0
   -1.0000         0         0    0.5000
         0         0         0    1.0000
```



Watch at https://youtu.be/o8BD3aQWW3I

Overshoot

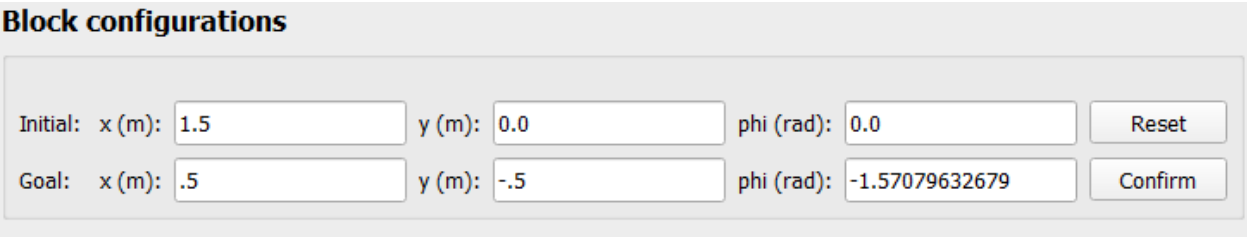
With the same starting positions, and the Kp = 5 and Ki = .1, the robot's motion is much less smooth. Although in the simulation the robot still completes the task, it is unlikely a physical robot would be able to achieve those joint velocities. Shown below is the Z twist error to highlight the overshoot.



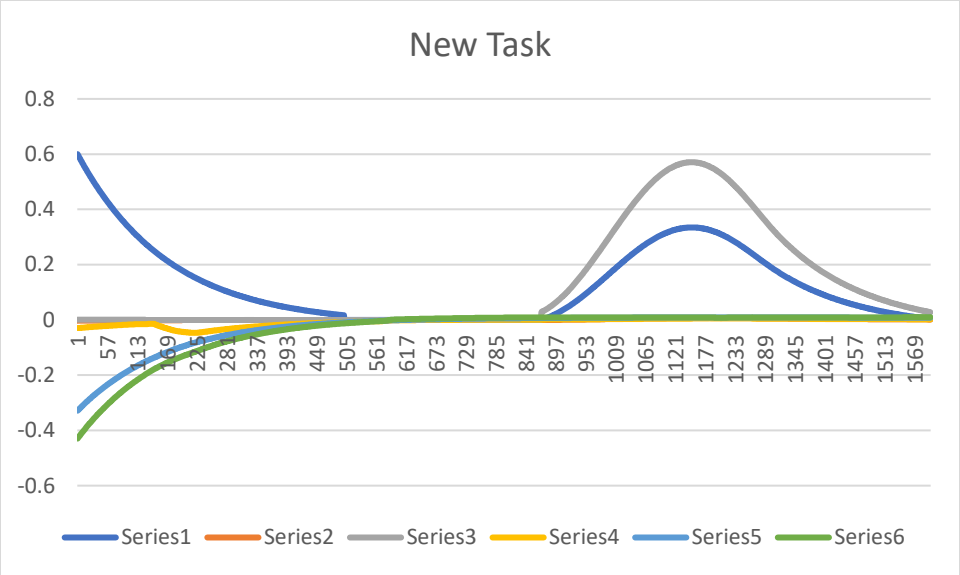
Watch at https://youtu.be/fOJ7GK--r_0

New task

With the same starting positions, and the gains back down to Kp = .6 and Ki = .01, the robot's new task is to grab a block further away and move it less far. The new block configuration is shown below.



By simply updating these configs in the beginning of the code, the robot follows this path and completes the task. This is done without updating any times needed between paths.



Watch at https://youtu.be/WTroTtBvOoY

Discussion

1. The integrator term in the feedback controller stops the position error from maintaining steady state error, but can cause overshoot or even instability if set too high. Overshoot occurs because the accumulated error hold a positive or negative sign for longer than the current actual error has that same sign.
2. I ran into this problem while troubleshooting. For the quicker movements, the lower max velocities do not allow the robot to follow the feedback controller's prescribed motion, so error increases. This error cannot be helped by the integrator term, because the commanded velocities are already maxed out. If the time allotted is increased, this effect can be mitigated.
3. Direct velocity control can be possible when controlling a stepper motor or when motor control amplifiers are used in velocity mode.
4. To implement torque control, you need information about the links moments of inertia, and other mass properties of the robot. The function would need to use functions like ForwardDynamics to calculate current mass properties that vary with position, and InverseDynamics to use those properties to fine joint torques. You would also use QuinticTimeScaling to have smooth 5^{th} order time derivates for smooth motion.