SERVO SIMULATOR

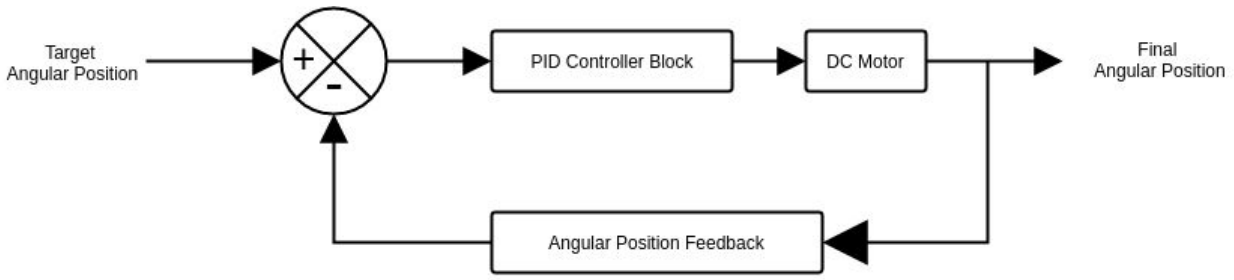
Simulation of a Servo Actuator as a Robot Joint

A servo actuator is an automatic self-correcting device that measures the error between the desired and the current position and sends it as a negative feedback to the system trying to minimize the same and converge to the former as soon as possible.

Introduction to Servo Motion Control

The closed-loop servo control over motion of a joint uses the error in desired and current positions to drive the transient response characteristics, steady state errors and the sensitivity to varying loads. An ideal controller would have lower settling time (meaning the gap between current and desired angular position is overcome as soon as possible), low steady state error (meaning the final angular position is almost equal to the desired angular position) and should be able to drive the system to desired goals regardless of changes in physical parameters like inertia(due to load), damping and friction.

Servo Working Mechanism:



Control System for Servo Control

The flowchart above shows the process of operation for the task. The user inputs a target angular position and the servo system tries to simulate the joint motion based on low-level commands generated from the PID Controller. It outputs the angular position that is fed back (negative feedback) to the comparator and this procedure continues until convergence.

Prototype of a 'servo simulator' unit that accepts a target position and returns a continuous cyclic stream of position values that show how a servo actuator may respond to the input target.

Approach:

The **servo simulator** is developed in **Python 3** language for the sake of simplicity of prototyping. It accepts a target angular position and tries to make the joint converge to that value using a PID controller.

The proposed **servo simulator** also has a few more options for control. It has options to simulate the **DC Motor** using the separately excited motor model or the armature controlled motor model**.** The user can pass the type of motor model as an optional command-line argument.

The proposed simulator has two options for initial simulation environment - either the user can pass the initial angular position of the joint as an optional command-line argument or the environment spawns the joint in a random angular position.

Command to run in the terminal:

$ python3 servo_simulator.py [target angular position] **-t** [type of motor] **-s** [initial angular position]

target angular position: float value between 0-90 degrees - Compulsory Input

type of motor : string value - Optional Input

Separately Excited Motor Model : 'sep'
Armature Controlled Motor Model : 'armc' (Default)

initial angular position : float value between 0-90 degrees - Optional Input

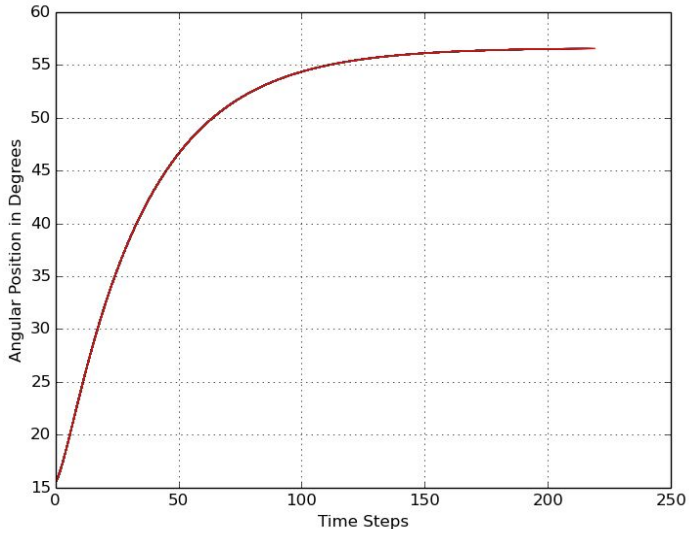
Simulator Output:

A continuous plot showing the simulator response, i.e. variation of the robot joint's angular position over time as it tries to converge to the target angular position. It also prints out a continuous stream of the angular position value for each time-step while trying to converge

Example Input 1:

$ python3 servo_simulator.py 56.5 -t armc -s 15.1

This command simulates the servo to reach 56.5 degrees starting at 15.1 degrees for an Armature Controlled DC Motor model.
The obtained output for this example:

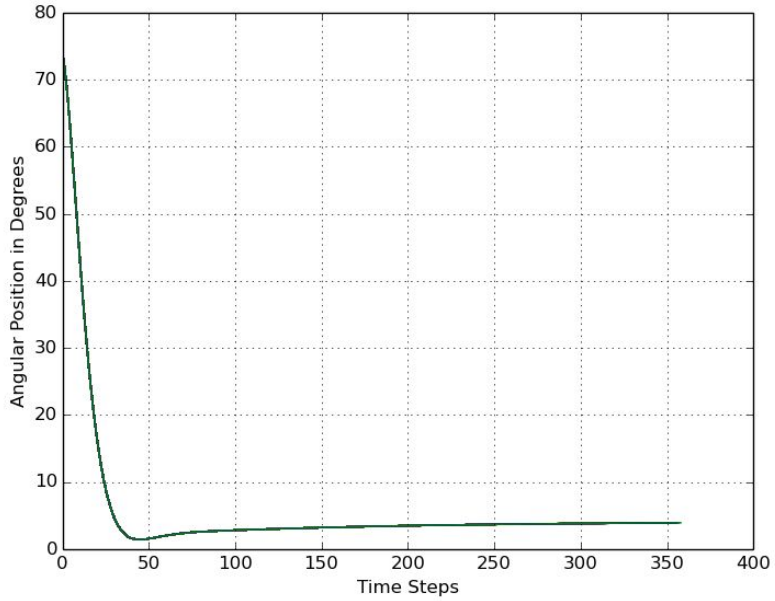


Target Angular Position: 56.5 Final Obtained Angular Position: 56.510330871227126
Example Input 2:

$ python3 servo_simulator.py 4.1 -t sep

This command simulates the servo to reach 4.1 degrees starting at a randomly generated angular position for separately excited DC Motor model.

The obtained output for this example:



Target Angular Position: 4.1 Final Obtained Angular Position: 3.90174364598

Description of the Simulator:

The Simulator implements servo control on a robot joint. It takes in the target angular position for the robot joint along with a few more optional command-line arguments mentioned above.

The Simulator has options to simulate two kinds of DC motor model.

1. **Separately Excited DC Motor**

   For a DC motor, the final velocity depends upon voltage applied, motor characteristics and the load. Any motor provides torque as the output that controls the velocity. It completely depends upon the electrical characteristics

   Rotor generated back emf : $E = K_e\omega$ ....(1)
   Torque for the motor : $T = K_t I$ ….(2)

   Where K_e and K_t are the electromotive force constant and the torque constant. I is the motor current which is also defined as $I = (V - E)/R$

   Finally, Torque $T = K_t (V - E) / R = K_t(V - K_e\omega) / R$ ….(3)

   Thus, the **torque** holds inverse linear relationship with the **angular velocity.** For the sake of modeling simplicity, the inductance induced voltage is ignored.

   Equation (3) is used to model the separately excited DC motor in the simulator which has predefined values for the constants.

2. **Armature Controlled DC Motor**

Using this model, we can deduce the relation between the motor angular velocity and the torque only using the mechanical characteristics of the motor.

The equations (1) and (2) still hold true for this model.

For an armature controlled DC motor, the Laplace equations for the dynamical system are given as:

$$T(t) = J_m * \frac{d\omega(t)}{dt} + B_m * \omega(t) \quad \text{....(4)}$$

$$T(s) = J_m * s * \omega(s) + B_m * \omega(s) \quad \text{....(5)}$$

Finally, the transfer function obtained is $\omega(s)$ = (1 / ($J_m s + B_m$) $* T(s)$ ....(6)

The Laplace inverse of Equation (6) is given as

$$\omega(t) = (1/J_m) * e^{-Bm/Jm} * T(t) \quad \text{....(7)}$$

Equation (7) is used to model the armature controlled DC motor in the simulator. Here, the values for J_m and B_m are replaced by $J_t = J_m + J_l$ and $B_t = B_m + B_l$ in the final model to accommodate the inertia and viscous friction arising due to the load along with the same values for the motor itself. We assume that inertia to be lumped.

The simulator also has an option to implement a gear-train for power transmission to the load that changes the load inertia and viscous friction by a squared factor. However, the PID values currently haven't been tuned for the same.

In both the motor models discussed above, the proposed simulator does not explicitly have a second order transfer function representing the relationship between the angular position and the torque, but a first order transfer function for the angular velocity and the torque. When the simulation is run, the system itself calculates angular position change by multiplying the output angular velocity with the simulation time-step size and updates the current angular position.

The default simulator time-step is 0.001 seconds which could be changed as desired.

Working of the Simulator

The simulator first creates an object of the Servo Simulator class and an object of the Motor model class within it. Thereafter, it reads the user defined command-line arguments that define the target angular position and other optional parameters like type of motor model to simulate and the starting angular position of the robot joint.

Here, the servo actuator is position controlled and runs on a PID controller but the actuator driving the joint is actually a DC motor which is essentially torque controlled. So, after the gathering all initial information, the simulator runs the PID controller to compute the "controlling torque" which is fed to the motor model of user's choice to simulate joint motion. This PID controller has different sets of tuned control values K_p, K_d and K_i for the two different motor models. It computes the "controlling torque" using the position error, rate of change of position error and integral of position error over time.

The proposed system assumes the PID control value to be direct value for the torque rather that passing it through a power interface/amplifier to get actual torque. This linear scaling step was ignored for the sake of simplicity. This torque generates angular velocity that is used to update the angular position for that time-step.

The **servo simulator** checks for the joint not exceeding the angular limits and clips the same to within the constraints in such a case.

The updated angular position is fed-back to the PID controller again and this cycle continues until convergence. The condition for convergence for the simulator is when joint angular velocity is less than 2 degrees/second and the steady-state angular position error is less than 0.2 degrees. This ensures that the controller makes the robot joint reach the target position and terminates smoothly.

Performance of the Simulator

The proposed simulator has satisfactory performance after tuning for both the motor models.

The Armature Controlled model was tuned perfectly and converges to a target angular position within acceptable time and limited overshoot.

The separately excited model was also tuned successfully and converges to a desired angular position, but with a comparatively higher convergence time.

A future task would to test the credibility of the simulator would be obtaining realistic values for the motor model and constants and check if the angular position still converges to the desired value. Currently, the constants used in the models are more of an educated guess. Incorporating the gear-train effects in the system and then observing the changes would also test the effectiveness of the **servo simulator.**

Another future task could be testing the controller for a trajectory, i.e., a sequence of time-distributed angular positions and angular velocities for the robot joint. Currently, the desired angular velocity is 0 for the set-point tracking task. Testing the controller on a trajectory would check how the system works for other real-life motion applications as well.