ME 8, Fall 2024-25 Report #1 - Software Architecture and Implementation
Ritali Jain and Diya Agarwal

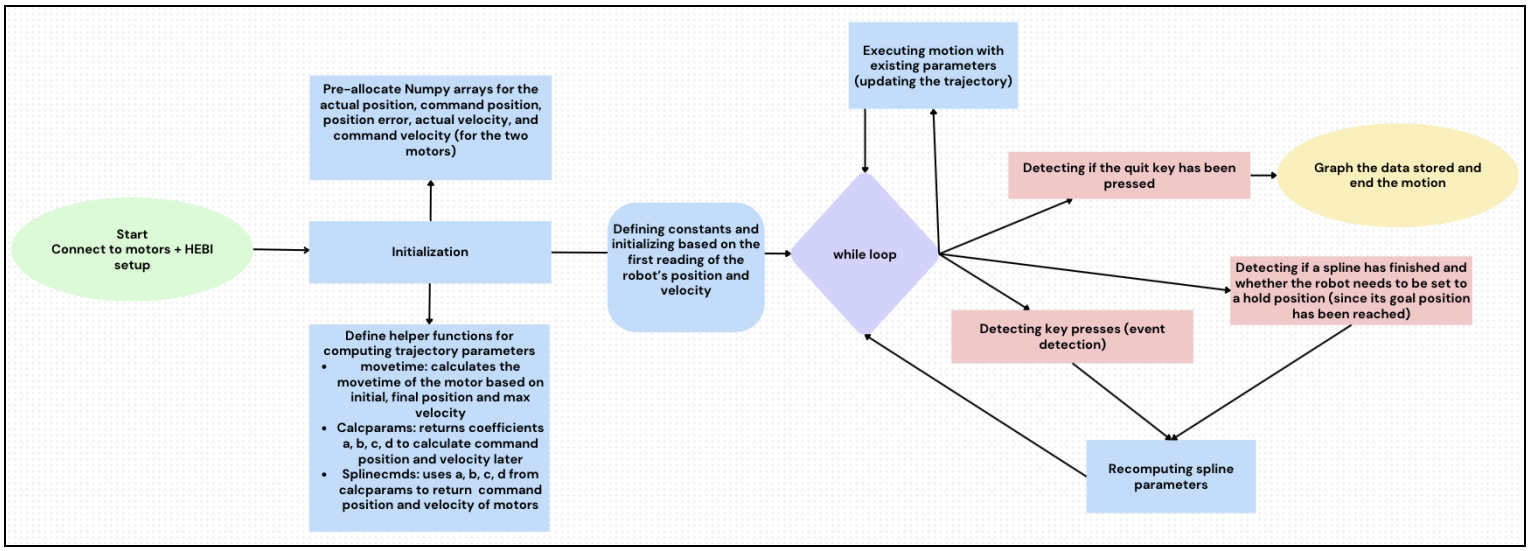
1 Introduction/Overview

The objective of the system is to control two motors based on user-controlled events, specifically, to control the motion of the robot (two degrees of freedom) based on key controls. We also want to ensure that the robot moves as quickly as possible (under its max velocity) as well as smoothly.

The system must move smoothly (continuous position and velocity), thus requiring the usage of splines to control the motion from one position to a goal position. We used cubic splines to define the parameters for our trajectories. The motor changes its position in a discrete fashion at time intervals of 0.01 s. The motor is user-controlled and must move to a specified location with keystrokes. We want our code to be error-prone (for example, undefined keys should not affect the behavior). Since we are also requiring smooth motions, our system must be able to change its motion in response to an event in the middle of a different motion.

Our software principles were to maintain the compactness and intuition of the code. Specifically, we organized our initialization, event-handling, and other components of the code to flow naturally and be readable, for example, by utilizing clear variable names. We also focused heavily on generalizability to ensure that our code is easy to refactor. We defined reusable functions for parameter computations, and we also relied on lists and condensing the code into vector operations using Numpy. This ensures a cleaner codespace, as there are fewer variables to keep track of, while still allowing us to control multiple degrees of freedom of the robot.

2 General Structure and Organization of the Code



Organizing our code allows us to handle events separately from the end of motion. In both cases, we are able to preserve the smoothness of the motion (continuity and differentiability), while providing generalizable code. We control the motion over discretized time, as per our requirement. At the end of every iteration of the while loop, we increment the time by dt = 0.01 seconds (i.e. 100 Hz) so that the motor knows where in the trajectory it should compute the new parameters, The line `feedback = group.get_next_feedback(reuse_fbk=feedback)`is a block in the motor that internally waits 10ms for reading the actual data. Obtaining feedback from the motor directly at a rapid rate allows the robot to detect any events that may occur at unknown times. Once an event is registered, the movetime helper method allows the robot to react to events (when a key is pressed, for example) at any time by recalculating the movetime for the spline at any given initial position and final position (and max velocity which is constant here). When a key is pressed, the movetime function is called and given the parameters of the new target position and the movetime is then computed.

3 Detailed Description of the Code Sections/Elements/Aspects

3.1 Signals and Parameters

Pcmd and Vcmd are continual 100Hz signals that are passed between the motor and the Pi. These are updated every cycle with readings from the motor. They are used in the computation of splines because they often define the initial conditions, so in that sense, they may be useful across more than just one iteration. We also get the Pact and Vact from the motor for plotting purposes by reading the feedback at each cycle (each cycle is 0.01 s or 100 Hz). Every 0.01 seconds, the pi reads the data from the motor and sends a new Pcmd and Vcmd based on the initial position of the motor and the desired position for the next interval. The signals for command positions and command velocity (Pcmd and Vcmd) are calculated using the spline formulas, shown in Table 1.

The following helper functions were implemented:

1. Movetime: calculates the movetime of the motor based on initial and final position as well as maximum velocity. This also adds an additional “waiting” time to reduce almost zero move time which reads in numerical instability of the velocity.

2. Calcparams: returns coefficients a, b, c, d based on initial time, final time, initial and final position, and initial and final velocity

3. Splinecmds: uses a, b, c, d from the calcparams function to return the command position and velocity of the motors.

Table 1: Variables used to compute position and velocity splines

Var.	Formula/Meaning	Where it is Used

a	$a = p_0$	To compute Vcmd and Pcmd
b	$b = v_0$	
c	$c = 3(p_f - p_0)/T_{move}^2 - v_f/T_{move} - 2v_0/T_{move}$	
d	$d = -2(p_f - p_0)/T_{move}^3 + v_f/T_{move} + v_0/T_{move}^2$	
p0	Motor's initial position	To calculate spline parameters, Pcmd, Vcmd
pf	Final desired motor position	
t0	Initial time of movement	
tf	Final time of movement	
Tmove	$Tmove = \lvert 3(p_f - p_0)/(2vmax) + v_0/amax \rvert$	
Pcmd	$Pcmd(t) = a + b(t - t_0) + c(t - t_0)^2 + d(t - t_0)^3$	Command position for the motors
Vcmd	$Vcmd(t) = b + 2c(t - t_0) + 3d(t - t_0)^2$	Command velocity for the motors

3.2 Logical Behavior / Event Handling.

In general, we handle different keystroke events that update the goal position of the motor in real time., which updates Pcmd. We use a dictionary, key_positions, to store the responses of the different key events, which just corresponds to changing the goal position of the motors.

Table 2: Event Handling Variables

Event	**Response**
'a', 'b', 'c', 'd', 'e', 'z'	Move the pan and tilt motors to specified radian positions, with 'z' defining the origin.
'q'	Quit the movement and plot the graphs
HOLD	When the total allocated time is about to run out, the final position is set to the initial position, and Vcmd is set to zero. This holds the motor in place.

Event handling ensures that the motor doesn't move after the allocated time runs out. Having two cases (able to move vs has to hold) allows the keystrokes to control the motion of the motor when time is remaining. During event handling, we use the current position and velocity and the initial position and initial velocity for the new trajectories being computed to preserve continuity. All three helper functions mentioned above are used when computing the new command position and velocity.

3.3 Initialization/Setup/Cleanup/Support Functionality

The order of the motors in the names passed into HEBI determines which motor is treated first/second when sending commands. We wrote our code such that the first value in the vector corresponds to the pan motor whereas the second value in the motor corresponds to the tilt motor.

Table 3: Initialized Constants

Constants	**Meaning**
Amax	The maximum acceleration approximated for the motors
Vmax	The maximum velocities of the pan and tilt motors
key_positions	Dictionary of response goal positions to keystrokes
offset	Since our motors do not completely align with the brackets, we overcame the hardware limitation with a software solution of adding an offset to the necessary command positions.

We collect Pact, Vact data at every iteration of the while loop and store all of our data in arrays. We then plot these data after the program has finished running (the plotted data consists of data that fit into a certain amount of storage, which is predetermined by our allocated lists).

Table 4: Graph Data Variables

Var.	**Formula**	**Where it is used**
N	$N = int(T / dt)$	Number of preallocated indices for storing the time, position, and velocity.
Time	$Time = [0.0] * N$	Keeps track of each time interval for the graphs
PAct	$PAct = np.zeros((2, N))$	The actual position of the motors, read through feedback commands. Uses Numpy arrays to save data for both motors. data saved for graphs.
PCmd	$PCmd = np.zeros((2, N))$	The command position sent to motors (Numpy array); data saved for graphs.
VAct	$VAct = np.zeros((2, N))$	The actual velocity of the motor(s), read through feedback

		commands.
VCmd	$VCmd = np.zeros((2, N))$	The command velocity sent to motors (Numpy array); data saved for graphs.

4 Performance and Tuning

In Figure 1 below, note in the system that the motion of the pan and tilt motors can be independent. Furthermore, notice how there are abrupt changes resulting from keypresses during a motion (i.e. an event occurs while the motor is not in a hold position) but the motion is still resolved continuously, as the graphs remain continuous. Also note that the max velocity is never exceeded.

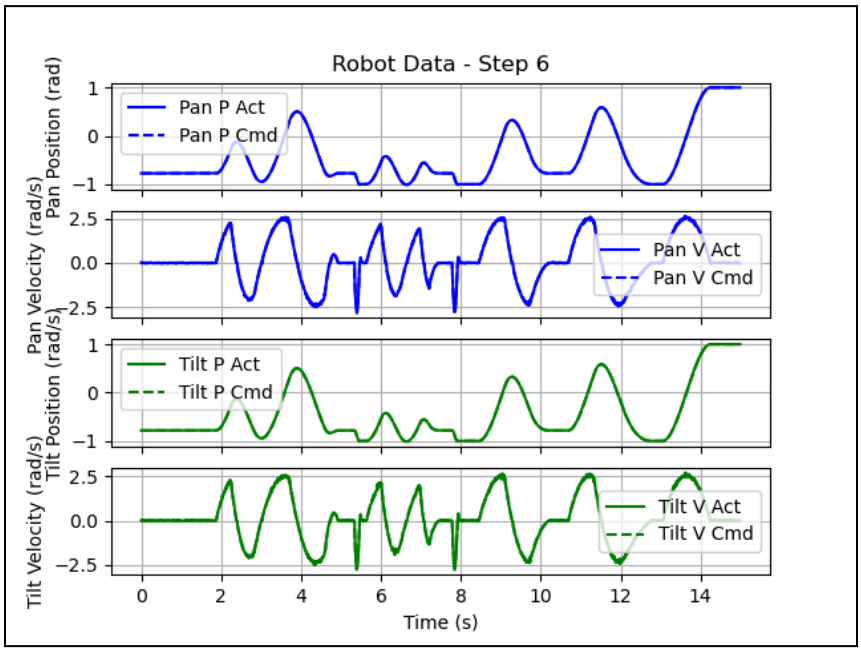


Figure 1: Pan and Tilt Motor Graphs. Blue represents the pan motor and green represents the tilt motor.

5 Features, Limitations, Options for Improvement

To ensure our code is readable and clean, we implemented numpy arrays to control both motors simultaneously without having to invoke different variables or events, allowing for interpretable and concise code. We also utilized a dictionary to handle our events, and we can freely append events to the dictionary without making the codebase messy.

The system as currently designed relies on max velocity for every computation of movement. In the case that the performance of the motor degrades over time, then the max velocity in the software may exceed the actual max velocity, which would cause noisy and jerky movements.

Our system explicitly relies on cubic splines, as it is the lowest degree polynomial that interpolates for multiple beginning and end conditions (for position and velocity).

In the future, we could also plot the positions of the two motors on the same graph to see how the motors are moving relative to each other. The same concept can be applied to the velocities of the two motors. Beyond the scope of this class, our code would generalize well for adding more degrees of freedom in motion, for example, if we wanted to add two separate cameras that move distinctly. Additionally, adding more key commands for different letters could be easily implemented through our dictionary of key press commands.