Individual Lab Report - 5

Autonomous Reaming for Total Hip Replacement

HIPSTER | ARTHuR

Sundaram Seivur

Team C:

Kaushik Balasundar | Parker Hill | Anthony Kyu

Sundaram Seivur | Gunjan Sethi

April 7 2022

Carnegie Mellon University

The Robotics Institute

Contents

1 Individual Progress **1**

2 Challenges **2**

3 Team Work **3**

3.1 Anthony Kyu 3
3.2 Kaushik Balasundar 3
3.3 Gunjan Sethi 3
3.4 Parker Hill 3

4 Future Plan **4**

1 Individual Progress

The sprint for progress review 4 was very demanding. In this sprint, I was incharge of completing the motion planning pipeline by writing a node to generate an arbitrary trajectory and publish the generated trajectory to a topic such that the controls subsystem can subscribe to it. For this, we decided to generate a custom message on ROS and consisted of the following:

- Trajectory message - header, joint names, joint positions, joint velocities and joint accelerations
- Pose array - header, x/y/z cartesian positions and x/y/z/w orientation in quaternion
- Point array - x/y/z cartesian velocities

This message helped in compiling all the necessary information for the controller to run its MPC solver. Fig.1 shows the message structure:

```
oem@sas:~$ rosmsg info arthur_planning/arthur_traj
trajectory_msgs/JointTrajectory traj
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string[] joint_names
  trajectory_msgs/JointTrajectoryPoint[] points
    float64[] positions
    float64[] velocities
    float64[] accelerations
    float64[] effort
    duration time_from_start
geometry_msgs/PoseArray cartesian_states
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose[] poses
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
geometry_msgs/Point[] cartesian_vel
  float64 x
  float64 y
  float64 z
```

Figure 1: Custom message structure

The cartesian states were not accessible directly, hence, I had to write a forward kinematics function to compute the end-effector position at each joint state in the trajectory. To compute the cartesian velocities, I found the derivative of the cartesian positions at two consecutive points.

I also wrote a node to generate a trajectory between the current state and an arbitrary point in the workspace. For this, I first read the current joint states from Gazebo and evaluated the end-effector's position. For the sake of simplicity and to help MPC converge faster, I generated

simple linear trajectories that would also mimic the reamer moving axially into the pelvis. Once the trajectory was generated using the Pilz planner, I populated our custom message with all the information. I wrote a simple publisher node to continuously publish these trajectories to the controller node at 10 Hz. I have written this node with modularity in mind such that any subsystem can access the required functions.

Finally, I collaborated with Anthony and Kaushik to integrate the planning and control subsystems. As part of this, we validated if our publisher subscriber nodes were communicating and if the controller node is able to parse into and access each waypoint in the trajectory.

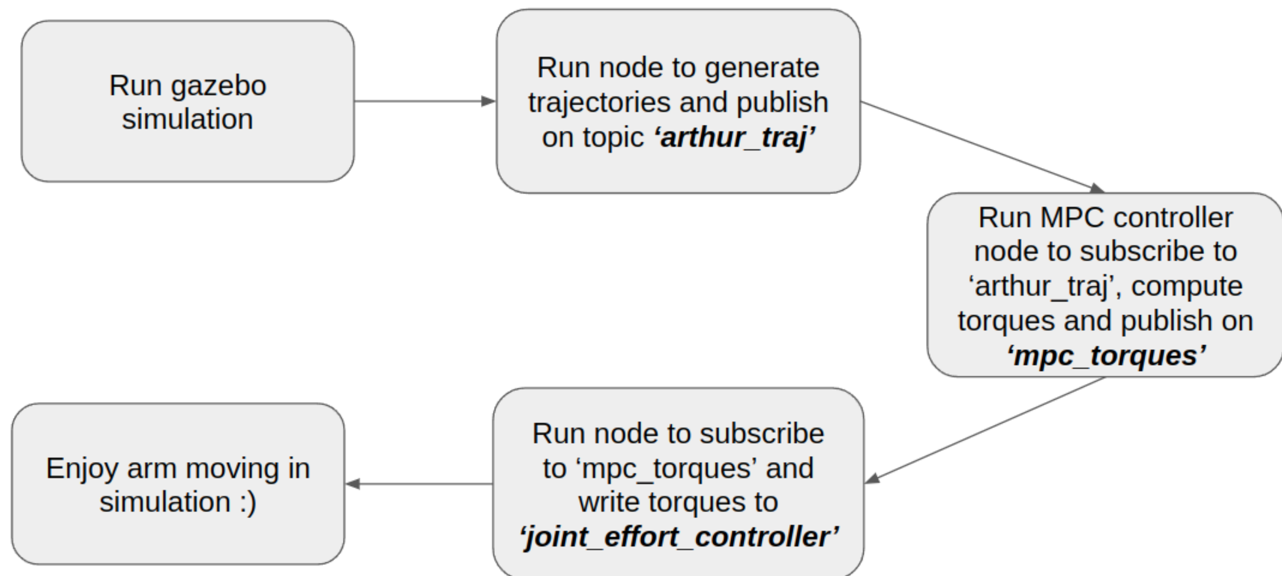

Figure 2: Planning and controls integration

2 Challenges

It was challenging to understand MoveIt’s class structure and hierarchy of information. Upon sending a motion planning request, it took me some time to study and understand the right way to access the motion plan response and parse into each waypoint. I had to do some trial and error, some break statements and a lot of print statements to check the output at each stage.
Since MoveIt! Plans trajectories in joint space, it was each to copy this to our custom message, however, there was no straightforward way to get the cartesian positions and velocity of the end-effector. The C++ interface has these functionalities, but I was coding in Python for quick prototyping. Hence, I wrote a forward kinematics function that acted as a client for the ‘compute fk’ service written by Kinova.
Due to programming in Python and constraining the planner to plan only linear trajectories, the planner takes a few seconds to compute the trajectory. This does not meet our performance requirements for dynamic compensation and needs some optimization.
As always, integration was challenging and proved to be difficult. The frequencies at which each

of the nodes were publishing and the rate at which MPC was solving was different. This leads to some lag in the solver.

3 Team Work

3.1 Anthony Kyu

Anthony worked on transferring the MPC code from offline simulation to online real-time simulation in Gazebo. This task involved writing MPC update functions as well as restructuring the code to be modular and more efficient. He then integrated the MPC code into ROS using RobotOS.jl, writing an MPC solver node to solve the optimal control problem, a simulation node for internal testing before integration, and a controller node to send torques to the effort controllers. Collaborating with Sundaram and Kaushik, Anthony then integrated the controller and MPC nodes with the trajectory planning nodes, Gazebo sensor nodes, and the Gazebo effort controllers, enabling real-time simulation testing of the controllers in Gazebo. Because the MPC had trouble performing well in real-time, Anthony also developed a PD Impedance Tracking controller in parallel to use as a fallback should the MPC not be viable by the SVD. Anthony also collaborated with Parker, providing tips on how to set up and wire the power distribution system.

3.2 Kaushik Balasundar

Kaushik worked with Gunjan to obtain the registration marker's tip pose using the market geometry and the pose of the probe center obtained from the camera. He then worked with her to obtain the pointcloud of the pelvis using this probe, and drafted a software architecture for the perception sub-system. After this, he worked on using the acquired pointcloud to develop a method to obtain the initial guess for registration, and further refined the pipeline to register the pointcloud of the acetabulum with the 3D CAD model of the pelvis. The registration was then evaluated quantitatively. He also assisted Sundaram and Anthony in the integration of planning and controls sub-systems by writing the effort commander interface.

3.3 Gunjan Sethi

Gunjan worked on extending the functionality of the perception pipeline and testing for robustness and reliability. She worked closely with Kaushik to integrate the new registration probe into the current pointcloud collection pipeline, obtain the probe tip pose and publish pointclouds at several frequencies. She developed user-input based sparse pointcloud collection and continuous dense pointcloud collection. Both functionalities were thoroughly tested. The point cloud collection pipeline was integrated and tested with the registration pipeline with the help of Kaushik. Further, she wrote test scripts to track multiple marker geometries as a proof-of-concept. Kaushik and Gunjan also discussed a revised software architecture for the perception and sensing subsystem.

3.4 Parker Hill

Parker redesigned the end-effector reamer to be shorter and more robust than his initial prototype. These designs were then 3D-printed and attached to the arm, allowing for the hardware system to be finalized for the Spring Validation Demonstration. He also worked on the motor

control PCB, ordering all the parts, soldering them to the PCB, and testing the PCB for efficacy. While the PCB is not finalized, he was able to extend the motor wires and connect them to a power supply to actuate the reamer head, allowing for testing of the hardware system to be conducted on the sawbone pelvis.

4 Future Plan

For our SVD, we would like to have Perception, Motion planning and Controls integrated to an extent. The perception subsystem would publish a transform (arbitrary) which would act as end point for reaming. I will port the planning pipeline from python to C++ so that we have lesser latency issues. I will spend a lot of time porting everything from simulation to hardware and have a basic set up running for all subsystems in reality. I would plan a trajectory from current state to an end point on the pelvis. Once I have planned a trajectory, I would orient the robot to the orientation of the endpoint and then move axially to reach the pelvis. I would also spend a lot of time integrating the necessary subsystems for SVD, making sure that we all subsystems running individually.