Anytime Compute Study

Trade-off profiling for ICP

Siddharth Gangadhar

Carnegie Mellon University

1 Motivation

In the context of control of autonomous robotic systems, sensing and actuation are tightly coupled in a feedback loop that consists of a state-estimator that draws inference on the robot's state from sensor data and a controller that uses the state information to generate appropriate control actions. Therefore, both computational time and the accuracy of the estimator can significantly impact the controller's performance. Typically, these estimation algorithms operate on a run-to-completion basis; they terminate after the best possible estimate of the state is obtained. And in the case of perception-based state estimators (localization), such as ORB-SLAM, the computational cost may incur delays that can severely degrade the control performance.

Anytime compute is a framework that can allow the controller to exploit the natural trade-off between the estimator's speed and accuracy during runtime by specifying a time of completion δ and error bound ϵ, in the form of a contract specified by the controller based on its immediate control objectives. Experimental results of which are presented in [2].

2 Iterative Closest Point

ICP aims to bring two pointclouds together via rigid-body transformations. Steps involved in ICP:

- Determine a set of correspondences between the two pointclouds. In this, we have used KDTrees and filtered out outliers by thresholding the max euclidean distance between closest neighbours.
- Compute the optimal translation and rotations required to bring the corresponding points closer in an iterative fashion.

2.1 Optimization Approach

We considered using the linearized SVD technique coupled with centroid translation, but we settled on using a sparse solver to compute the optimal translation and rotation at each iteration. we first define the following parameterized transformation:

$$T(x;\xi) = R(\xi)x + t(\xi) \tag{1}$$

Where $\xi = \begin{bmatrix}\theta & t_x & t_y\end{bmatrix}^T$. Using this we defined the cost-function as follows:

$$J(\xi) = \sum_{j\in\Omega} \|q_j - T(p_j;\xi)\|^2 \tag{2}$$

Where q_j are points from the target pointcloud and p_j are from the source pointcloud. We can then use a nonlinear solver such as Levenberg Marquardt to solve for the optimal pose ξ^* as follows:

$$\xi^* = \arg\min_{\xi} \sum_{j\in\Omega} \|q_j - T(p_j;\xi)\|^2 \tag{3}$$

3 Experiments and Profiling

3.1 Test setup

With `slam_toolbox` we created a map of my lab area and then used a rosbag to record a test trajectory of the vehicle using the GPU-accelerated particle filter as shown in Fig 15.

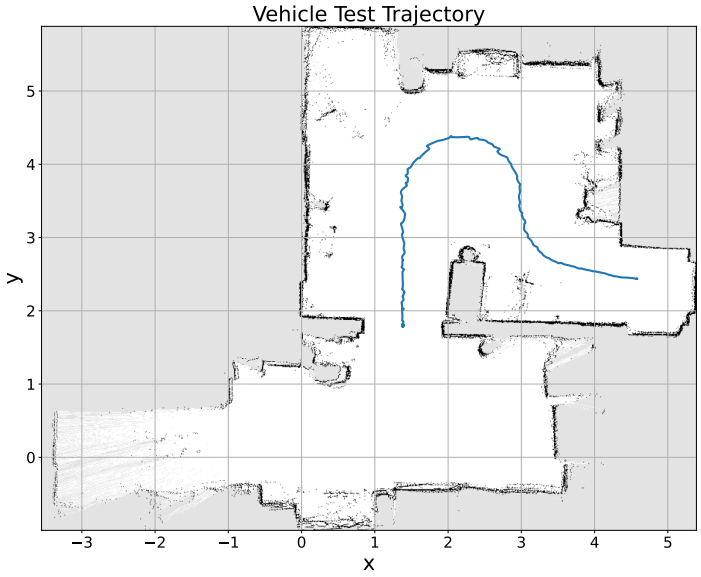

Figure 1: Vehicle Trajectory from Particle Filter

we recorded the `/scan` topic along with the odometry pose topic, `/pf/pose/odom`, from the particle filter.

3.2 Testing the ICP Algorithm

Fig 16 shows the raw laserscan pointcloud data that was recorded during the test.

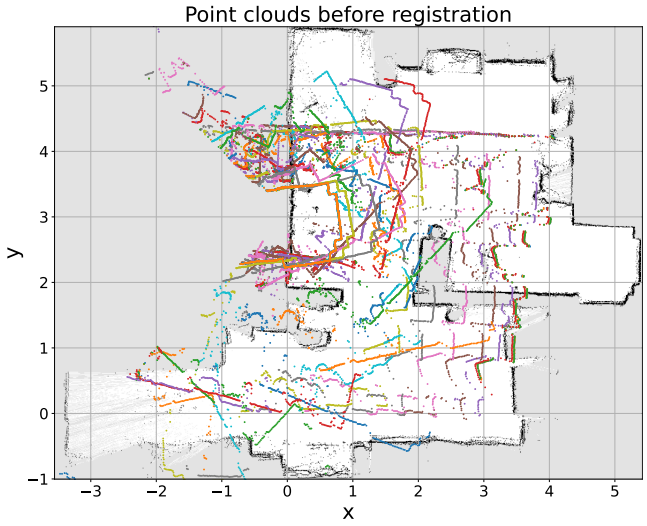

Figure 2: Raw Pointcloud

Next, we use the pose estimate from the particle filter to initialize the pose matrix T_0 for the ICP. In a loop, for each laser scan, we run the ICP and obtain the relative transformation between the current (target) and next (source) frame T_{rel}. we then compute the updated pose at the next time step as follows:

$$T_{k+1} = T_k T_{\text{rel}} \tag{4}$$

To verify if the obtained poses are correct, we plotted the transformed pointclouds on the map and, sure enough, as seen in Fig 17 we see that the pointclouds do match map features.

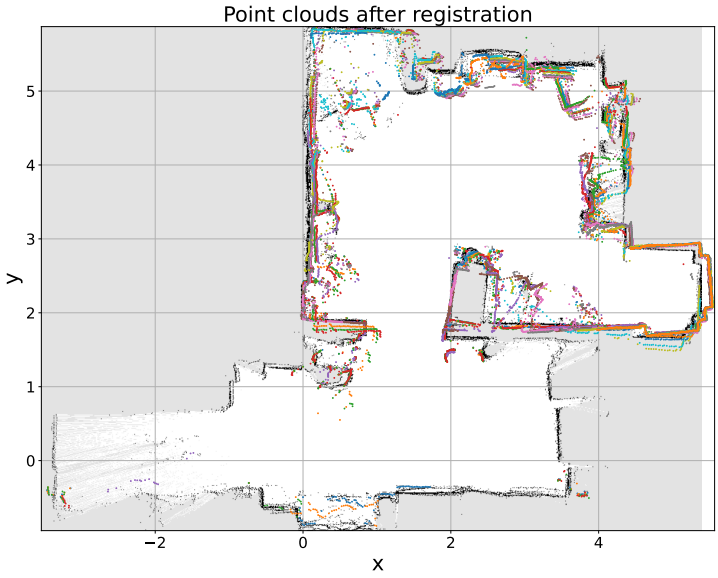

Figure 3: Pointcloud After ICP Registration

3.3 Comparing ICP and Particle Filter

Here, we am using the pose estimate from the Particle Filter (4000 max particles) as "ground truth". Next, we needed to find a relevant parameter, or *knob*, such that varying them results in changes in the computation time (δ) and the quality (ϵ) of the overall output of the ICP. The obvious choice in this case is the maximum number of iterations, `MAX_ITERS`, allowable before

termination. Fig 11 compares the vehicle states for the particle filter and ICP with different `MAX_ITERS` values.

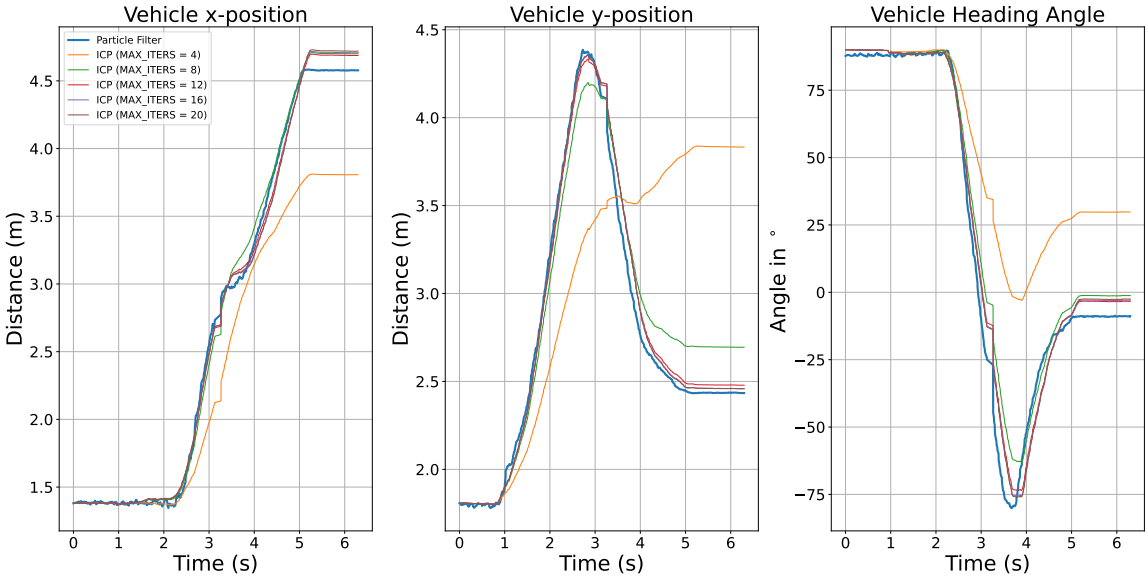

Figure 4: ICP v/s Particle Filter Pose Estimation for different max iterations

In a similar way, Fig 12 compares the estimated vehicle trajectory on the map.

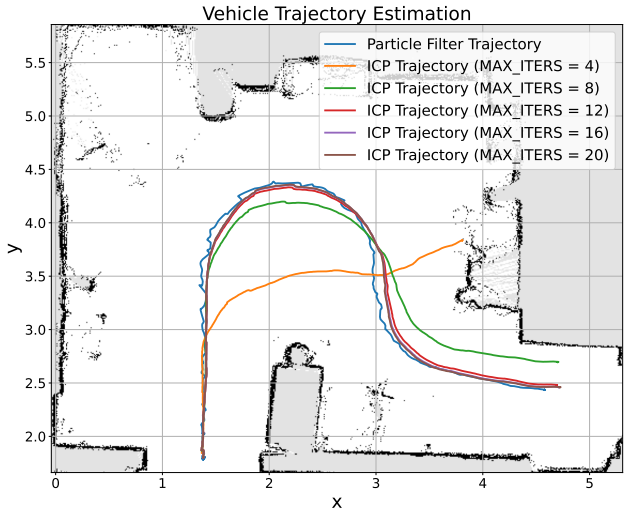

Figure 5: ICP v/s Particle Filter Trajectory Plot for different max iterations

We see that increasing the `MAX_ITERS` parameter causes the ICP's estimated pose to converge to that of the particle filter's.

3.4 Profiling the ICP

Measuring the average computation time δ for each value of `MAX_ITERS` and the mean square error between the resulting pose estimate the ground truth results in the profiled delay-error curve shown in Fig 13

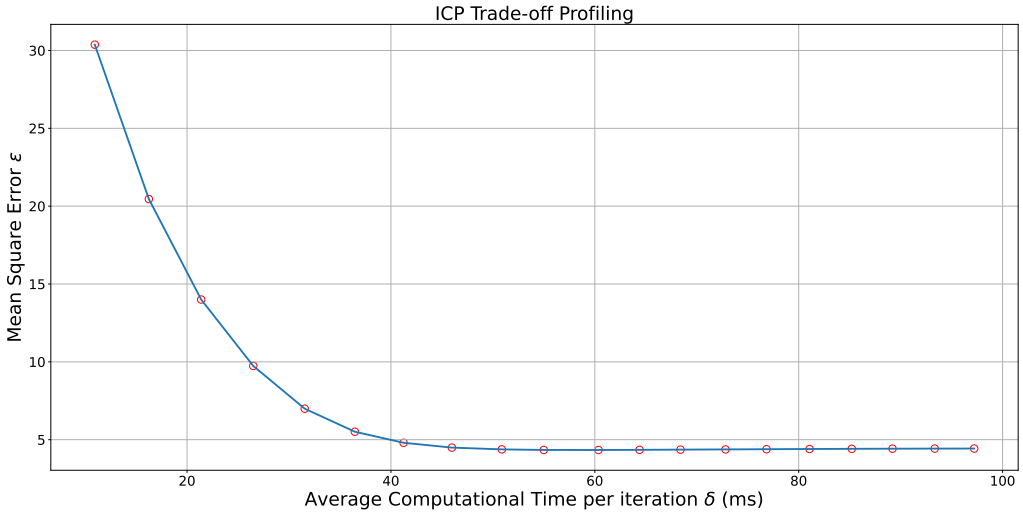

Figure 6: ICP Trade-off Curve

3.5 Profiling the Particle Filter

Since we already had the rosbag data, we also wanted to profile the particle filter itself. For the ground truth value, we use the same pose estimate from the case with 4000 max particles. And for the *knob* we used the max particles. The obtained delay-error curve is shown in Fig 14

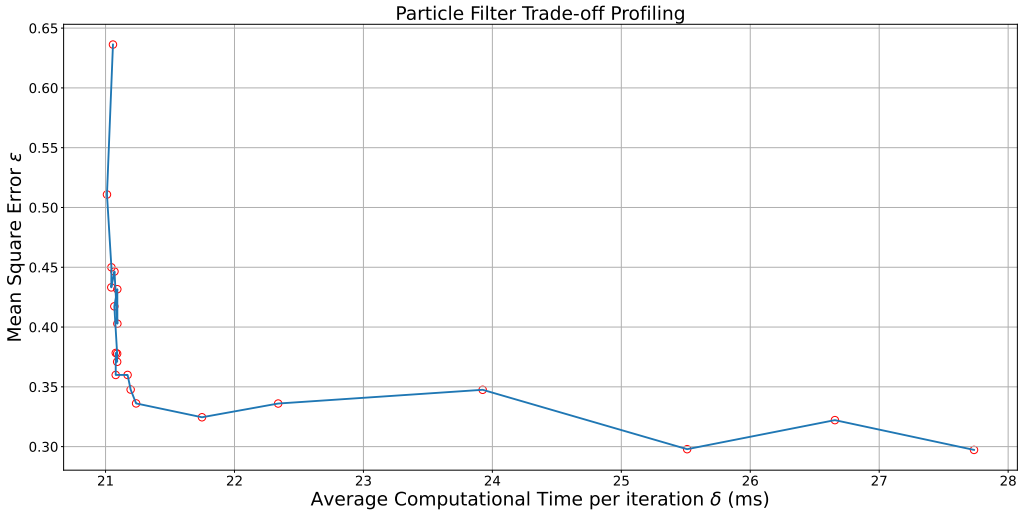

Figure 7: Particle Filter Trade-off Curve

4 Particle Filter Profiling

4.1 Attempting to 'interrupt' the Particle Filter Computation

Previously, my attempt at profiling the particle filter was to vary the maximum number of particles and compare the accuracy v/s computation delay of the recorded trajectory. However,

as Zirui pointed out, this is not necessarily the correct way to go about doing so. Zirui suggested that in the implementation of the code, in between resampling, the code uses the current odometry along with the previous state estimate to update particle motion and that varying this time interval using a message filter could induce an accuracy v/s delay trade-off.

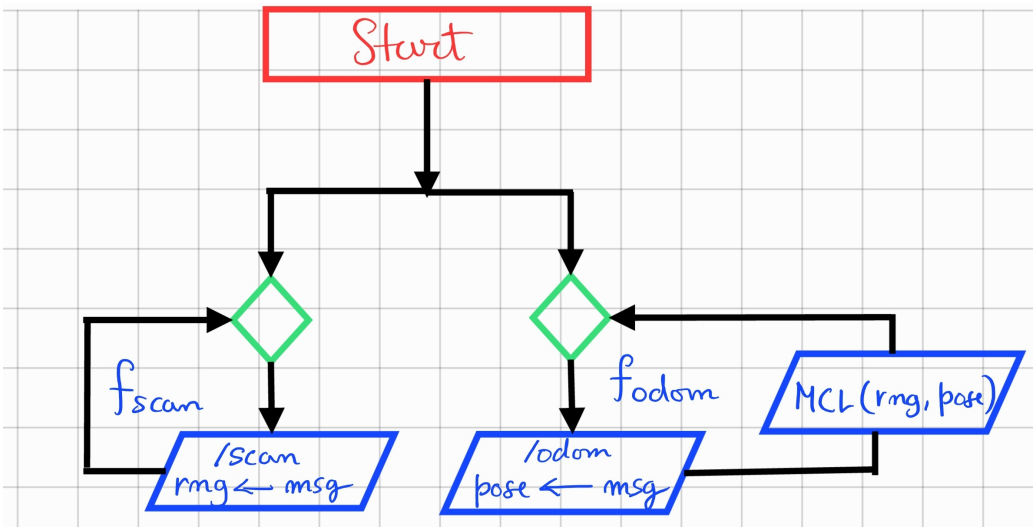

Figure 8: Original Particle Filter Code Structure

Fig 15 shows the flow of the original particle filter code [3]. There are two (main) subscribers, one for the laser scanner and the other for odometry. In this implementation the odometry callback function also calls the MCL update function that handles resampling, motion update, and measurement correction steps as well as the inferred pose calculation.

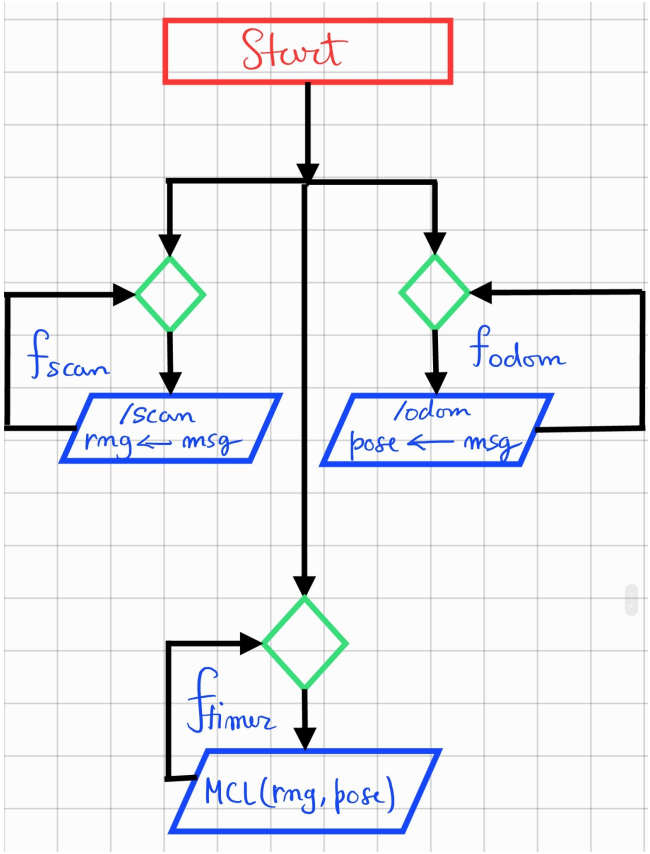

Figure 9: Partially-Interruptable Particle Filter Code Structure

Fig 16 shows the flow of the modified particle filter algorithm. Here, we have tried to emulate

the behaviour of an interrupt service routine via a wall-timer with varying callback periods. Here, the MCL updates the proposal distribution asynchronous to the sensor message rates. For update rates much faster then the sensor messages, the particle filter uses the same odometry and scan measurements until they are updated, and from what we saw experimentally, this leads to a degradation in performance of the inferred pose. The accuracy improves until the callback frequency matches the sensor message frequency. Any slower will significantly ruin pose estimates.

4.2 Experimental Results

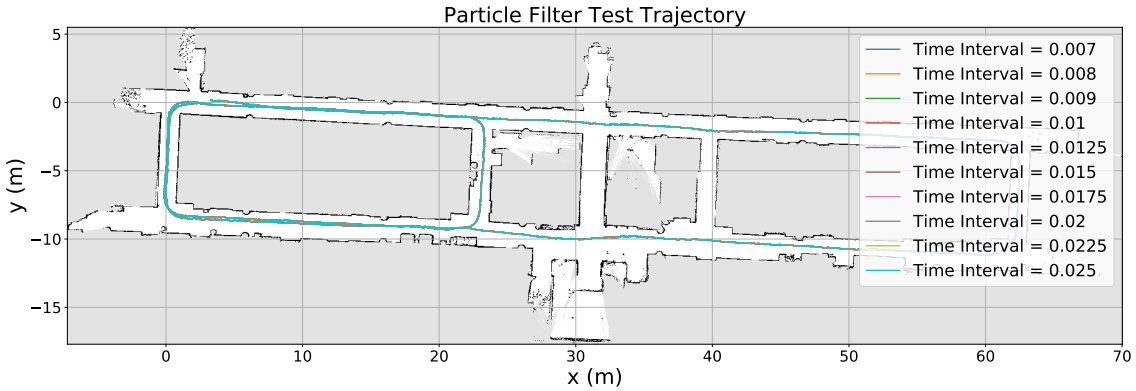

Figure 10: Particle Filter Test Trajectory

Fig 17 shows the vehicle trajectory obtained during a long experiment. The trajectory is roughly 210m long. This is required to discern small variations in different trajectory that would otherwise be lost in shorter trajectories.

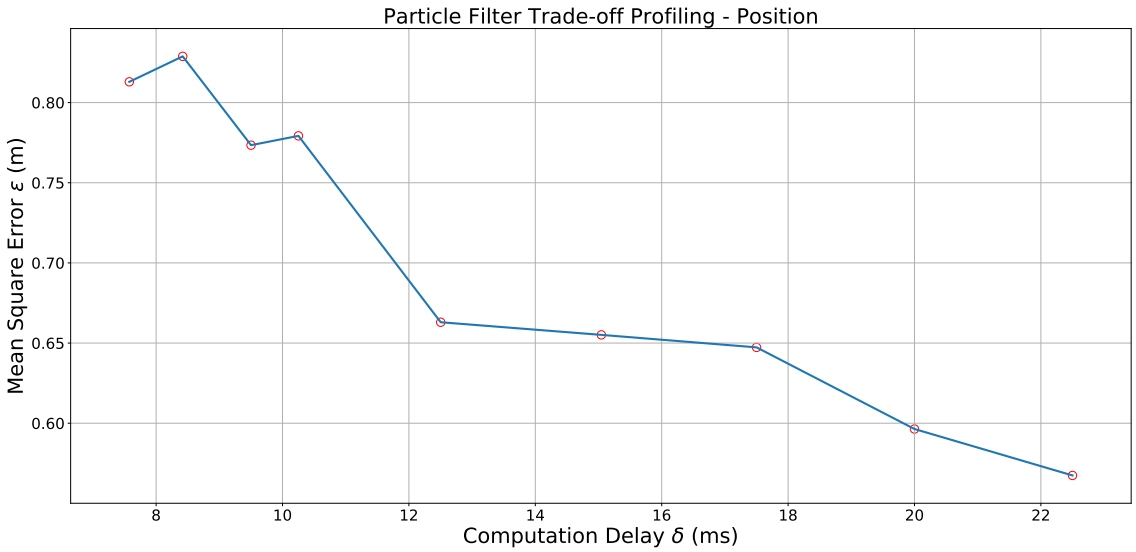

Figure 11: Particle Filter Trade-off Curve Position

Fig 11 shows the mean square position error between the each of the obtained PF trajectories v/s the PF trajectory that is synchronous to the sensor message.

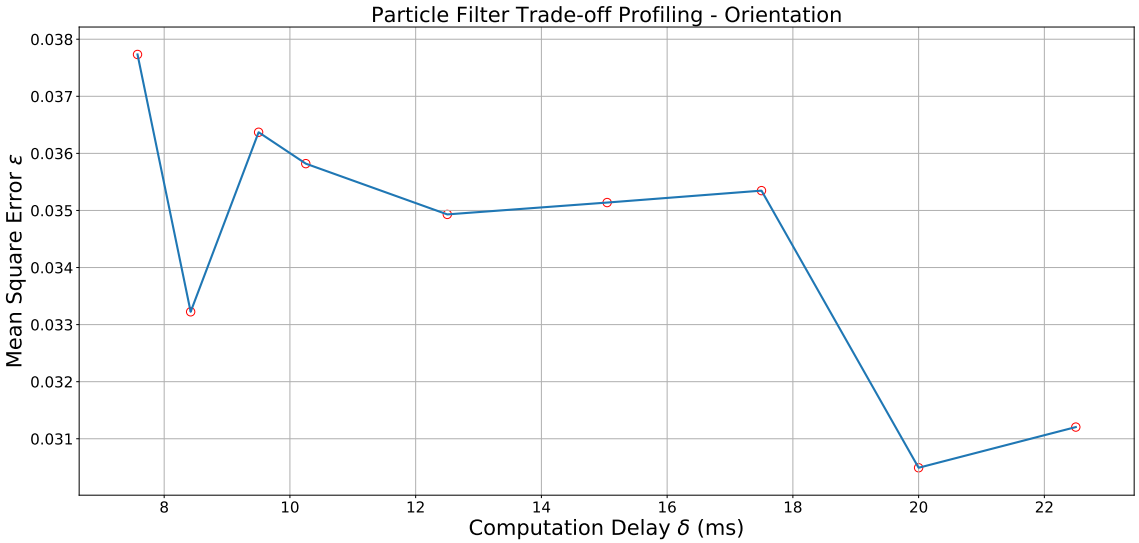

Figure 12: Particle Filter Trade-off Curve Orientation

Similarly, Fig 12 shows the error in orientation between the heading quaternions. Some notable features; the position error seems to behave monotonically, while the orientation is not so monotonic. we'd have to analyze this more closely.

5 *Incremental* updates to ICP Profiling

Previously, we was using a linear approximation to the roto-translation operation and a sparse QR solver. This week, we experimented with a few nonlinear solvers and also worked on the point to line metric with a quadratically constrained QP solver. we've used the same experimental data we obtained in the previous week's work to study these variants. The point to line ICP was inspired by [1].

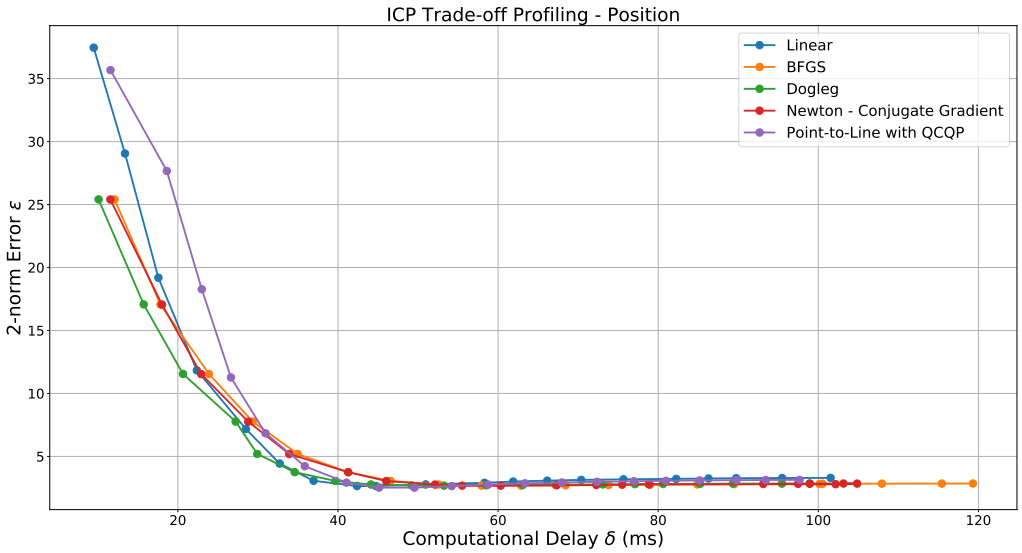

Figure 13: ICP Trade-off Curve Position

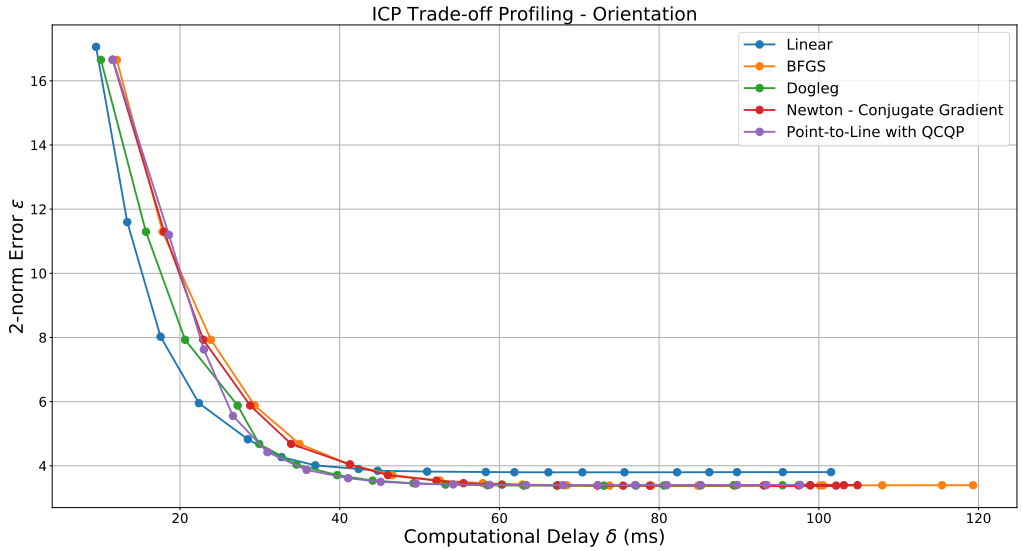

Figure 14: ICP Trade-off Curve Orientation

Fig 13 and 14 show the ICP error v/s delay trade-off for position and orientation respectively. For the point to plane ICP, we did not use the fast correspondence search using the jump-tables, we will work on that over the next week. This explains why it is among the slowest methods here.

6 Shared Memory Framework

Reading a bit into shared memory and interprocess communication system in Linux, led me to the XSI Shared Memory Facility `<sys/shm.h>` and Interprocess Communication Access Structure `<sys/ipc.h>`. This framework allows for the creation of a common shared memory file with a unique key. One can allocate a fixed number of bytes to this shared memory block and obtain a pointer to the address of the first byte, as shown below:

```
key_t key = ftok("/path/to/file/shmfile", 65); // key_t ftok(const char *pathname,
    int proj_id)

int shmid = shmget(key, 1024, 0666 | IPC_CREAT); // shmget(key_t key, size_t size,
    int shmflg)

int *ptr = (int*) shmat(shmid, (void*)0, 0); // void *shmat(int shmid, void *shmaddr
    , int shmflg)
```

This system works between two ROS Nodes and run asynchronously and can be accessed at anytime and at any point during execution by simply dereferencing the pointer associated with the attached shared memory file. It is possible to use pointer arithmetic to use more than one shared variable, which is useful for handshaking.

7 ICP Interruption and Profiling

7.1 Initial Efforts on building an Interruption System

My initial ideas for a contract based system, wherein the controller interrupts the estimator midway to receive an immediate estimate, had a lot of issues during implementation. The first

issue we encountered was the need for some handshaking between the controller node and the ICP node. Suppose the controller wishes to interrupt the ICP at time 40ms from the start of estimation, but the ICP completes it within 30ms; there needs to be some way for the controller to acknowledge that estimation was completed and stop the interrupt signal and reset its timer. My initial thought was to use a simple binary signaling system; 0 indicates that the ICP is currently running and 1 indicates that it has completed. However, based on experiments, the duration for which this signal would remain at 1 was ephemeral at best and the controller node would rarely ever acknowledge this. Its clear that to signal completion of estimation needs a more instantaneous event trigger. we therefore chose to go with an edge based signalling system. The ICP would toggle a binary variable on and off each time it completes estimation. This was reliably detected by the controller.

Algorithm 1 Controller Node Callback (*/icp_state*)

```
prev_state ← 0
procedure CALLBACK(msg)
    Create subscriber to /icp_state
    Start Timer Thread, timeout= T_int
Timer Thread - Timeout System
procedure TIMER(T_int)
    while T_elpsd < T_int and !clear_flag do
        if *(ptr + 1) ⊕ prev_state then
            prev_state ← *(ptr + 1)
            return
    clear_flag ← True
    *ptr ← *ptr ⊕ 1
```

Algorithm 2 ICP Node Callback (*/scan*)

```
prev_state ← 0
procedure CALLBACK(msg)
    Publish integer as /icp_state
    From scans get pointclouds (p, q)
    Run ICP(p, q)
    Publish Transform as /icp/odom
procedure RUN ICP(p, q)
    Build Tree
    for i ∈ {0, N_iters − 1} do
        Iteratively Compute Transform...
        if *(ptr) ⊕ prev_state then
            break
```

For the timer interrupt procedure, we used a thread to countdown to a specified timeout period and toggle the interrupt signal to the ICP. However, because my system relies on polling from the ICP completion signal, clearing the timer thread is highly unreliable. This lead to race conditions and other undesirable effects that made this framework unusable in practice.

7.2 Revised Contract based System

After discussing this Zirui, we both figured that a better approach to this would to make use of a timing contract/budget that the controller would specify to the ICP. This seemed to work much better. The controller passes a time to completion that the ICP must return an estimate within. To build this, we first had to understand the computational time-delay v/s the number of ICP iterations. Once we had obtained this data, we fit a function to use as a look-up table in the ICP Node. This is shown in Figure 15 below:

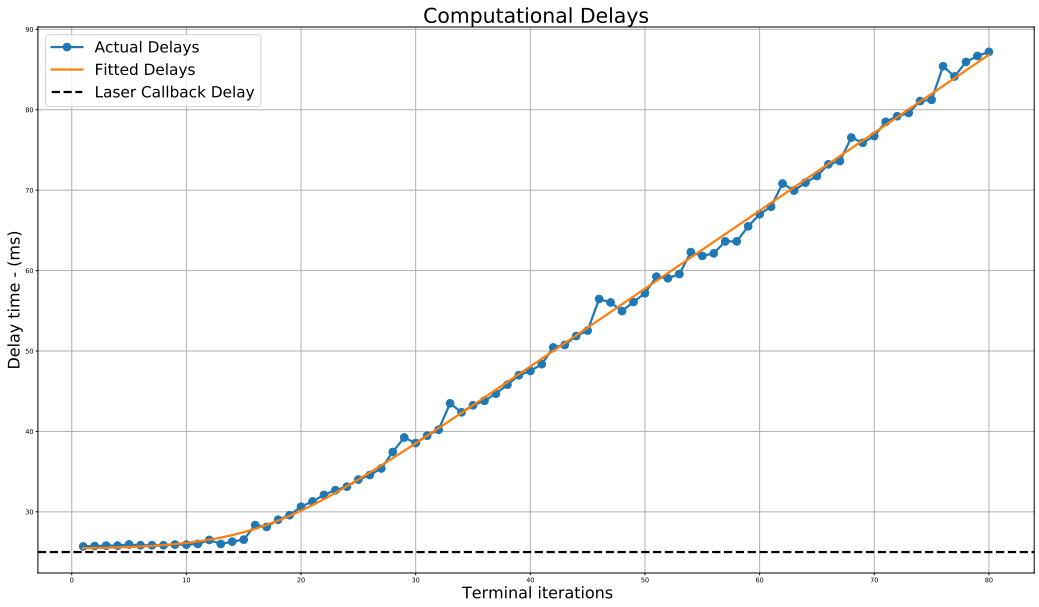

Figure 15: Computational Delay

Then using this we was able to profile the ICP as shown in Figures 16 and 17:

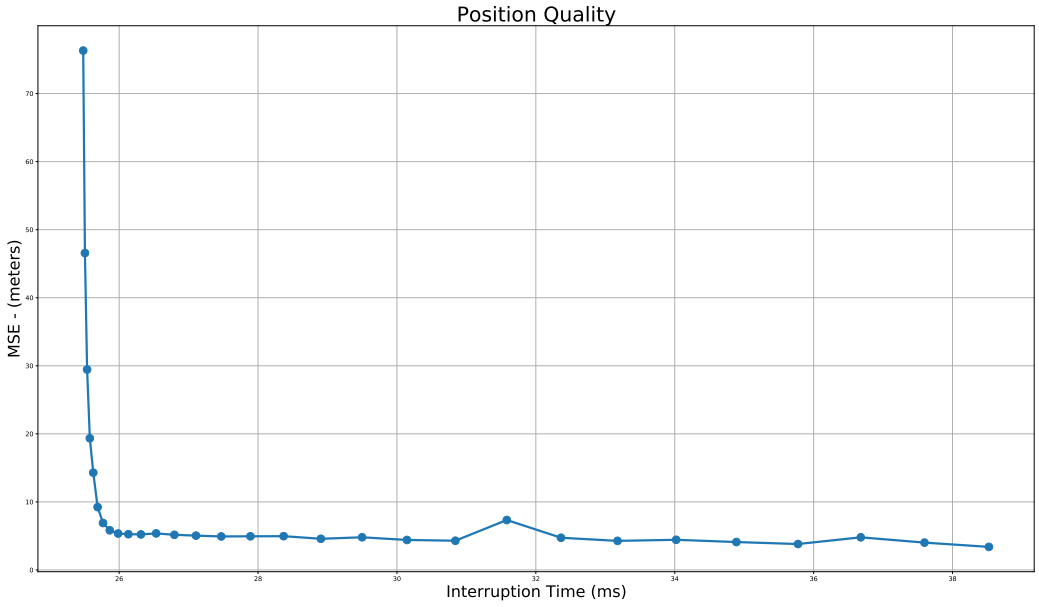

Figure 16: Position Quality v/s Delay

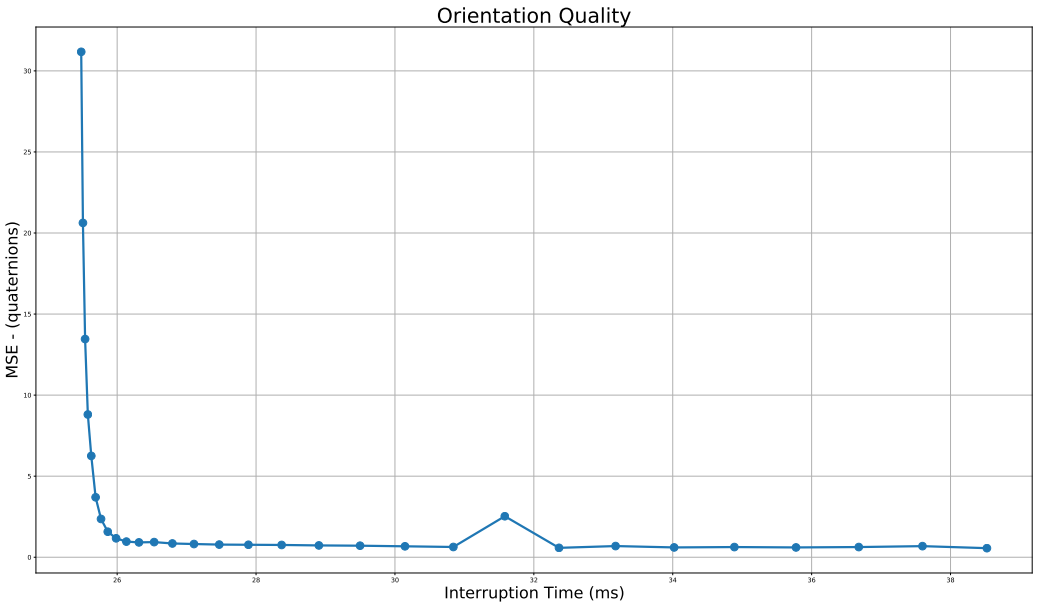

Figure 17: Orientation Quality v/s Delay

References

[1] Andrea Censi. An icp variant using a point-to-line metric. In *2008 IEEE International Conference on Robotics and Automation*, pages 19–25, 2008.

[2] Yash Vardhan Pant, Houssam Abbas, Kartik Mohta, Rhudii A. Quaye, Truong X. Nghiem, Joseph Devietti, and Rahul Mangharam. Anytime computation and control for autonomous systems. *IEEE Transactions on Control Systems Technology*, 29(2):768–779, 2021.

[3] Corey Walsh and Sertac Karaman. Cddt: Fast approximate 2d ray casting for accelerated localization. abs/1705.01167, 2017.