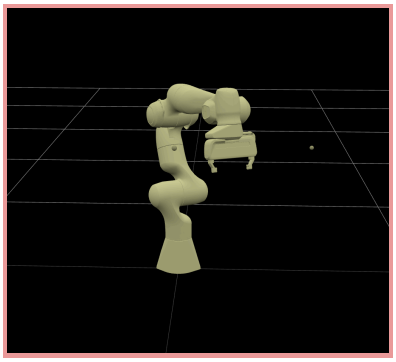
DK80 & ZL51

Trajectory Optimization

Problem and Motivation: Trajectory optimization aims to produce smooth, collision-free, dynamically feasible motions for robotic manipulators, but methods like TrajOpt are sensitive to initialization and often get trapped in poor local minima. For a 7-DOF Panda robot, a single random initial trajectory may not provide enough structure for TrajOpt to converge to a high-quality solution—especially in cluttered environments with strict velocity and acceleration limits. This project implements TrajOpt for the Panda, adds spherical obstacles to the environment, and evaluates how well TrajOpt performs under random initializations.

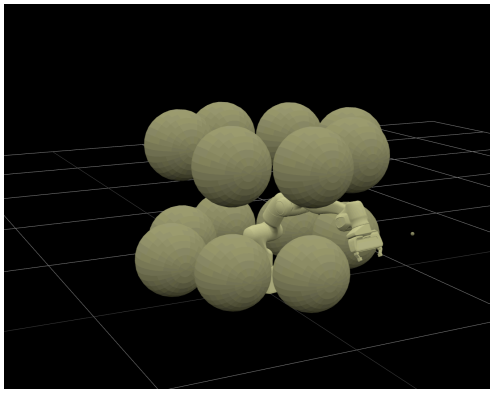
Modern robots require motions that are safe, smooth, and efficient, which geometric planners alone cannot provide. Sampling-based planners like RRT-Connect quickly find feasible but non-smooth paths; trajectory optimization can refine these paths, but only if given a good starting point. By generating multiple initial solutions using RRT-Connect with different random seeds, this project investigates whether better initialization can improve TrajOpt's reliability, reduce planning time, and yield higher-quality trajectories. Understanding this relationship is important for real-world robotic systems where consistent, collision-free motion is essential.



Approach:

Prework

1. The first step was cloning vamp and grabbing the panda folder for the robot arm
2. Cloning the Tesseract python example and getting that to work
3. Creating txt file with obstacle locations

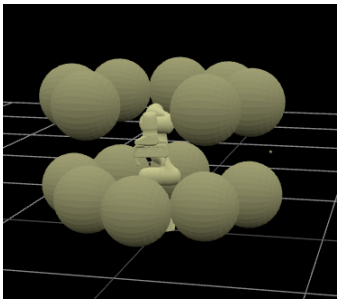


PART(A) Approach

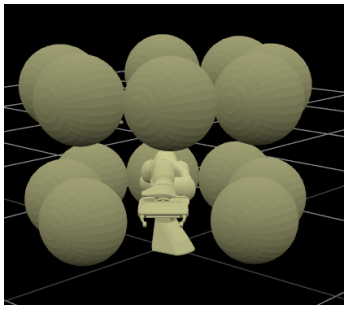
To begin Part (a), we started by replicating the provided example in tesseract and clearing the entire environment to ensure that the localhost interface would run correctly in the browser. This was done in a standard windows environment and not on wsl or in container. Once the base setup was working, we organized the Panda robot files within our project's directory structure and attempted to load them into the environment. During this process, we discovered that the provided Panda URDF was tailored for ROS and included group definitions that were incompatible with our standalone setup. After removing these group tags and making the necessary adjustments, the URDF loaded successfully.

Next, we converted the obstacle description from the given `.txt` file into an object format compatible with the environment, allowing us to properly render it alongside the robot. With these components in place, we were able to visualize the Panda arm and spherical obstacles as shown in the reference figure.

After establishing the environment, we extracted key constraints relevant to the robot and workspace, such as velocity limits, jerk bounds, and start/end joint configurations. These parameters provided the foundation for all subsequent experiments and served as baseline values for comparing the performance and behavior of different trajectory optimization strategies. We also established set start and goal points in the environment: there are two gaps present in the rings of spheres, one gap on the top level and one gap on the bottom. We set the start and goal points to be the two gaps in the spheres as shown below.



Start



Goal

The final stage involved implementing TrajOpt within the Tesseract environment, which proved to be the most challenging portion of the assignment. We explored several variations of the optimization setup in an attempt to obtain collision-free solutions.

Initial Planner

In the first version of the TrajOpt-based planner, we implemented a minimal end-to-end pipeline that goes from environment setup to trajectory visualization, with as little tuning as possible beyond what is necessary to get a working solution.

We started off with a more naive motion planning strategy. We would use a set amount of waypoints and create a linear interpolation in joint space between the start and goal. We would then clip the waypoints to the robots joint

limits to ensure feasibility. After this initial interpolation, we would then increase the density so it was suitable for TrajOpt.

To make the problem easier for this first variant, collision handling is configured to use collision costs instead of hard collision constraints, with a small collision margin buffer. This makes the optimizer more tolerant of initially colliding trajectories and lets it "push" the path out of collision instead of failing outright.

Since TrajOpt outputs only the sequence of joint configurations (no timing), the result is passed through a time-optimal trajectory generation step. Approximate velocity, acceleration, and jerk limits for the Panda are specified, and timestamps are assigned to all waypoints. The optimized trajectory is then flattened for analysis in order to calculate the trajectory length, maximum jump between waypoints, and whether or not the motion is reasonable.

We experimented with the number of waypoints to see the effect that it would have on the motion planning as a whole:

Number of Waypoints	2	5	10	30	50
Planning Time (s)	0.316	0.449	2.336	9.298	9.455
Trajectory Length (rad)	4.079	6.104	8.489	13.927	20.65
Waypoints in optimized trajectory	6	21	46	146	246

From these results we observed that planning time, trajectory length, and number of waypoints in the optimized trajectory increases as the number of waypoints increases. This clearly makes sense, as an increase in the number of waypoints in the interpolation would result in more calculations and a path that winds around the obstacles more. Something interesting to note is that it seems that both planning time and trajectory length do not increase linearly, but rather something akin to logarithmically. For planning time, this can be seen from how the planning time drastically spiked up when increasing the number of waypoints from 10 to 30, but the same increase was not observed when increasing the number of waypoints from 30 to 50. Similarly, the rate of increase in trajectory also falls off when increasing from 10 to 30, and 30 to 50. This seems to imply that at a certain point, for this type of naive motion planning strategy and environment, diminishing returns occur when increasing the number of waypoints.

After finding success with this planner, we tried to increase the complexity by setting collisions as a **constraint** and not as a cost: this resulted in the planner being unable to find a feasible solution. This is to be expected, as a simple linear interpolation in this dense environment most likely wouldn't be able to produce correct results. This now brings us to our second approach.

Second Planner

For our second planner, we extend the basic TrajOpt pipeline into a more robust "plan–then-verify" framework. The main goals are to produce trajectories that are explicitly collision-free under hard constraints, and to independently validate the resulting path using both discrete and continuous collision checking. To achieve this, we made a multitude of changes.

We changed the straight line interpretation into a jittered line in the space. We also introduced randomness into the jittering so there would be noise in the intermediate waypoints between the start and goal. This encourages the optimizer to explore slightly different initial paths while still being structured and feasible. We also increased the density of the waypoints which hypothetically would give TrajOpt more options to adjust the path in the hard-constraint phase

And most importantly, we changed the structure of the optimization into two phases:

1. Phase 1
    a. We allowed TrajOpt to path plan with soft collision costs, instead of having hard collision constraints. This would essentially allow TrajOpt to create an outline for a collision-free path. This process was expedited by having a collision margin buffer of around 2 cm. This allowed us to be tolerant while still having a smooth path that is pushed away from obstacles. This was basically our initial approach with an additional factor of a collision buffer.
2. Phase 2
    a. A second TrajOpt profile was created where collisions were considered as a constraint instead of just a cost. This profile would build upon the path that was found by Phase 1 and refine it such that the path would satisfy the hard collision constraints.

Unfortunately, despite tuning various hyperparameters, such as the number of waypoints, margin-width, seed etc, we were unable to utilize this approach to find a collision free and reasonable path for the robot. All of the hyperparameter combinations allowed us to pass Phase 1 without issue, but would fail to successfully complete Phase 2. This framework was specifically designed to see if we could find a completely collision free path: there is no point in allowing Phase 2 to allow collisions, as then we could just take the result from Phase 1 and call it a day. Furthermore, Phase 1 is the same as our initial approach, so it did not make sense to experiment with hyperparameters when allowing collisions as a soft cost. With this in mind, we then designed our third approach

Third Planner

For our third planner, we stepped back from the strict hard-constraint formulation of the second approach and instead focused on making TrajOpt “practically good” rather than “formally perfect.” The goal here was to reliably obtain smooth, short, and visually reasonable trajectories in a cluttered environment by improving the quality and diversity of initial seeds, biasing TrajOpt strongly toward smooth, low-cost solutions with collisions treated as a soft cost, and cleaning up the resulting path with collision-aware shortcutting.

For initialization, we generalized the jittered straight-line idea from the second planner into a reusable “seed generator.” Each seed trajectory is built as a straight line in joint space between the start and goal, with small Gaussian noise added to the intermediate waypoints and all joints clipped to their limits. This seed is then interpolated with a relatively low density, keeping the optimization problem smaller and smoother. The planner is written to support multiple random seeds, where each seed produces a candidate trajectory and the best one is selected based on total joint-space length. In other words, rather than relying on a single initial guess, this planner is structured to search over a small family of reasonably different but still structured initial paths.

Inside TrajOpt, we moved away from hard collision constraints and returned to treating collisions purely as a soft cost, but with significantly more emphasis on smoothness. We enabled velocity, acceleration, and jerk smoothing (when supported by the bindings) and increased the corresponding coefficients to penalize rapid changes in joint motion. On the collision side, we tuned the safety margin, buffer, and cost coefficient so that near-colliding states are strongly penalized but do not make the problem infeasible.

After TrajOpt finds a solution (and after attempting time parameterization when the waypoint types allow it), we apply an additional post-processing step that neither of the previous planners had: randomized **shortcutting** with collision checks. The idea is to repeatedly take two distant points along the trajectory and test whether the straight joint-space line between them is collision-free, using the discrete contact manager and sampling several intermediate configurations. If it is safe, we replace the entire subsegment with the shorter direct connection. This process tends to remove unnecessary detours introduced by the optimizer, resulting in a shorter and cleaner path. Finally, we filter out near-duplicate consecutive waypoints to remove tiny jitters.

We then experimented with varying the jitter scale, safety margin, margin buffer and the TrajOpt collision coefficient. All of the hyperparameters were run independently to judge the effect that these parameters had on the path as a whole.

Jitter Scale	Planning Time Mean (s)	Planning Time STD (s)	Trajectory Length Mean (rad)	Trajectory Length STD (rad)	Best Planning Time (s)	Shortest Trajectory Length (rad)
0	0.989	0.395	6.500	0	0.780	6.500
0.01	0.716	0.073	7.036	1.107	0.576	6.130
0.02	0.809	0.167	7.064	0.549	0.541	6.344
0.05	0.869	0.203	8.027	0.735	0.576	6.731
0.1	1.034	0.259	11.388	1.832	0.631	8.985

From these results we see that increasing jitter scale generally makes both planning time and trajectory length grow, especially at 0.1 where the mean trajectory length jumps sharply. However, a small amount of jitter (around 0.01–0.02) appears beneficial: the mean planning time actually drops compared to 0 jitter, and the shortest trajectory overall is found at jitter 0.01. This suggests there is a sweet spot where a little noise helps escape bad straight-line seeds, but too much jitter makes the trajectories unnecessarily long and more expensive to compute.

Safety Margin	Planning Time Mean (s)	Planning Time STD (s)	Trajectory Length Mean (rad)	Trajectory Length STD (rad)	Best Planning Time (s)	Shortest Trajectory Length (rad)
0.005	0.842	0.1995	7.7844	0.8908	0.5591	6.8744
0.01	0.8686	0.1935	8.027	0.7346	0.5898	6.7311
0.02	1.0832	0.2093	8.9949	1.4702	0.7293	10.1418
1	1.615	0.5414	5.0823	1.91	1.0833	4.3284

From these results we see that increasing the safety margin from 0.005 to 0.02 generally increases both planning time and average trajectory length: the optimizer is penalizing proximity to obstacles more aggressively, which tends to push the path farther around them and makes the problem harder. The 0.02 setting is the worst of this range: it has the highest mean planning time and a significantly longer mean trajectory.

Interestingly, the very large safety margin of 1.0 behaves differently. It has the longest mean planning time overall (the optimization is clearly harder), but it also produces the shortest paths on average and the globally shortest trajectory (4.33 rad). This suggests that with such a large margin, the planner is forced to commit to a route that stays far from obstacles, and once it finds that route, it is quite direct. In practice, this indicates a trade-off: small margins (0.005–0.01) give moderate planning times and reasonable lengths, while very high margins can yield safer and shorter paths at the cost of significantly more computation and variability.

Margin Buffer	Planning Time Mean (s)	Planning Time STD (s)	Trajectory Length Mean (rad)	Trajectory Length STD (rad)	Best Planning Time (s)	Shortest Trajectory Length (rad)
0.002	0.8757	0.1986	8.027	0.7346	0.5825	6.7311
0.005	0.8655	0.1947	8.027	0.7346	0.57	6.7311
0.02	0.8738	0.2004	8.027	0.7346	0.5763	6.7311
0.1	0.8619	0.1962	8.027	0.7346	0.5788	6.7311

From these results we see that varying the collision margin buffer has almost no effect on the planner's behavior. The mean trajectory length and its standard deviation are identical across all tested values, and the shortest trajectory found is also the same in every case (6.7311 rad). Planning time means differ only slightly (within ~0.015 s), which is on the order of noise rather than a meaningful trend. This suggests that, for this particular environment and TrajOpt setup, the margin buffer parameter is operating in a "flat" regime: small changes do not significantly change the optimization landscape, and other factors (seed randomness, solver noise) dominate the observed variation.

Coefficient	Planning Time Mean (s)	Planning Time STD (s)	Trajectory Length Mean (rad)	Trajectory Length STD (rad)	Best Planning Time (s)	Shortest Trajectory Length (rad)
1.0	0.3229	0.0301	6.4044	0.3154	0.2706	6.2145
10.0	0.8717	0.1909	8.027	0.7346	0.5759	6.7311
20.0	0.993	0.2119	8.3232	0.6926	0.5806	7.0832
50.0	1.1703	0.1983	8.9602	0.5164	0.852	8.7328

From these results we see that increasing the collision cost coefficient consistently makes the problem more expensive to solve and produces longer trajectories on average. Planning time rises from about 0.32 s at coefficient 1.0 to about 1.17 s at 50.0, while the mean trajectory length grows from ~6.40 rad to ~8.96 rad. The shortest trajectory found also degrades steadily as the coefficient increases.

This behavior matches the intuition that a larger coefficient punishes proximity to obstacles more aggressively, pushing the optimizer to take wider, more conservative routes that are longer and require more computation. However, for this environment, the "gentler" setting (coefficient 1.0) actually gives the best combination of speed

and path quality: it plans roughly 3 times faster and finds noticeably shorter trajectories while still respecting the modeled costs.

These were the three main approaches that we took in regards to trying to use only TrajOpt. We were unable to successfully find a feasible path that took in collisions as a hard constraint. Interestingly, when we reduced the obstacle radius from 0.2 to 0.1, TrajOpt began producing clean, feasible solutions without penetrating the spheres. This implies that the main issue was the density of the obstacles with respect to the size of the panda robot.

PART(B) Approach

While reviewing the VAMP README, we noticed that the framework provided a straightforward setup workflow for WSL. Based on this, we decided to complete Part (b) using Ubuntu 22.04 for a cleaner and more consistent build experience. The foundational environment setup closely mirrored the process from Part (a), but integrating VAMP's rttconnect module into Tesseract was significantly simpler by comparison. With the recommended dependencies and configurations in place, the VAMP components integrated smoothly, allowing us to focus primarily on analyzing performance rather than troubleshooting environment issues.

First, we keep the same Panda + obstacle environment and define fixed start and goal configurations. Instead of asking TrajOpt to solve the entire problem from a naive straight-line initialization, we use VAMP's RRT-Connect planner to generate multiple collision-free joint-space paths between the start and goal. Each of these VAMP paths is then converted into a Tesseract CompositeInstruction and densified via interpolation, giving TrajOpt a smooth, physically plausible initial trajectory that already respects the obstacle layout.

Next, we run TrajOpt separately on each of these candidate paths, using softer collision costs rather than hard collision constraints, and then apply time-optimal trajectory generation (with explicit velocity, acceleration, and jerk limits) to obtain dynamically feasible, time-parameterized trajectories. For each run, we compute metrics such as total trajectory length, duration, and planning time, and select the "best" solution (in our case, the shortest joint-space path). We also verify that the final trajectory remains within joint limits and has reasonable waypoint spacing, and we visualize the result in the Tesseract viewer for qualitative inspection.

This two-stage approach—sampling-based planning followed by local optimization—solves our earlier problem in two key ways:

> Better initialization: VAMP provides collision-free, topologically sensible paths that start TrajOpt in a "good basin" of the cost landscape, reducing the chance of the optimizer cutting through obstacles.
>
> Multiple candidates: By generating several VAMP paths with different random seeds, we explore different homotopy classes in configuration space and let TrajOpt refine each one. This increases robustness and makes it much more likely that at least one optimized trajectory is both collision-free and smooth.

In summary, VAMP gives us robust global exploration, and TrajOpt provides local smoothing and time-parameterization. Together, they overcome the local-minima issues we observed in Part (a) and produce higher-quality, collision-aware trajectories.

Optimizing before interpolating vs no optimization:

VAMP Method	Joint Space Length (rad)	Planning Time (s)	Waypoints	Duration (s)
Raw (No Simplification)	37.4766	6.2199	1646	56.2216
Simplified/Optimized	20.0509	14.2923	836	28.7802

The comparison between raw and simplified VAMP paths reveals a crucial insight into how the quality of the initial trajectory influences the final results of TrajOpt optimization. When VAMP's simplification is enabled, the optimized trajectories improve dramatically—joint-space motion drops by 46.5%and execution time is reduced by nearly 50%, even though the simplification step adds extra planning time. This shows that the structure of the initial path, not just its collision-free validity, strongly affects optimization performance.

The main reason for this improvement is that simplification removes redundant detours and shortens path segments where collision-free shortcuts exist. This reduces the raw 298–444 waypoints down to 168–229, and even after interpolation, the simplified paths still contain far fewer points. With fewer waypoints, TrajOpt solves a much smaller optimization problem, allowing it to converge more effectively and avoid being trapped in high-dimensional local minima. The simplified paths also provide better geometric "hints," giving the optimizer a clearer starting point that encodes more efficient routes through cluttered environments.

Overall, the results show that path quality matters as much as path validity. Although raw VAMP paths are collision-free, their density and conservativeness restrict TrajOpt's ability to reshape the trajectory, resulting in long, inefficient motions. In contrast, spending a small amount of extra time on simplification yields trajectories that are almost twice as efficient to execute. For practical robotics systems—especially ones with limited computation—this suggests that performing path refinement before optimization is a highly effective strategy.

Trajectory length and the planning time with and without obstacles

Part(a) TrajOpt

Environment	Cartesian Path Length (meters)	Planning Time (seconds)	Joint Space Length (radians)
Empty (no obstacles)	1.7093	0.0910	4.2942
With obstacles	3.7220	0.5090	6.9669

The results clearly show that adding obstacles has a major effect on both the trajectory quality and the computation time when using random initialization in TrajOpt. The Cartesian path length increases by about 118% (from 1.71 m to 3.72 m), which makes sense because the robot has to take a much longer detour to avoid the 14 spherical obstacles while staying collision-free. Planning time also goes up significantly—by roughly 459% (0.091 s to 0.509 s). This slowdown happens because TrajOpt must perform more collision checks and solve a harder optimization problem when obstacles are present.

The joint-space path length shows a similar pattern, increasing by around 62% (4.29 → 6.97 radians), which aligns with the longer Cartesian path. Even with these increases, TrajOpt still achieves a 100% success rate in both environments (5/5 runs), showing that the optimizer is reliable under both conditions. Also, despite the slowdown, the absolute planning times remain well under a second, meaning this method is still feasible for near real-time applications. However, the ~5× increase in time with obstacles is something that should be taken into account when working with systems that have strict timing requirements.

Part(b) Vamp RRTC

Environment	**Cartesian Path Length (meters)**	**Planning Time (seconds)**	**Joint Space Length (radians)**
Empty (no obstacles)	4.0268	3.3143	12.9269
With obstacles	10.6034	6.1701	37.4766

The VAMP RRT-Connect + TrajOpt pipeline behaves very differently from the random initialization approach. When obstacles are present, the Cartesian path length increases dramatically by 163% (4.03 m → 10.60 m), and the joint-space motion grows by almost 190%. This happens because VAMP tends to generate very conservative paths with large clearance around obstacles. These paths are already quite long before TrajOpt begins refining them, and since TrajOpt is a local optimizer, it mainly smooths the trajectory rather than searching for a fundamentally shorter one.

Planning time also increases from 3.31 s to 6.17 s, but the more important observation is that the absolute planning times remain much higher than in the random-initialization case. This is largely due to VAMP producing extremely dense trajectories—often 500 to 1600 waypoints—which create a much larger optimization problem for TrajOpt. With so many waypoints, TrajOpt focuses on local smoothing and collision constraint satisfaction, but it becomes much harder for it to restructure the overall shape of the path.

Even though the paths are longer and the computation time is higher, the pipeline still achieves a 100% success rate. This reliability comes from the fact that VAMP always provides a collision-free initial guess, which is valuable in scenarios where guaranteed feasibility is more important than speed or path optimality.

Overall, the results show a clear trade-off: VAMP+TrajOpt offers very robust performance, but at the cost of longer trajectories and significantly higher planning times. This makes the method more suitable for offline or non-time-critical planning, where reliability is prioritized over efficiency.

Conclusion

In this project, we explored how trajectory optimization behaves under different initialization strategies and planning pipelines for a 7-DOF robotic arm operating in a cluttered environment. Our experiments showed that TrajOpt alone is highly sensitive to its initial seed: simple straight-line or random trajectories can lead to long, inefficient motions or poor convergence, especially when obstacles are present. Adjusting parameters such as jitter, safety margins, collision costs, and waypoint density helped improve reliability, but these methods still struggled to consistently avoid local minima. This demonstrates a core challenge in trajectory optimization: while it is powerful for smoothing and refining motion, its performance is fundamentally limited by the quality and structure of the initial trajectory.

Introducing a global planner through VAMP RRT-Connect significantly improved robustness by providing collision-free seeds that guide TrajOpt toward feasible regions of the configuration space. Moreover, we found that the quality of the initial global plan matters just as much as its feasibility: simplified VAMP paths consistently produced cleaner, shorter, and smoother optimized trajectories than raw ones. Overall, the results highlight that high-performance robotic motion planning often requires a combination of global search and local optimization.

Difficulty of Assignment: 8
Time Spent: 64 hours