League 1: Rapid Reaching Competition Report

Jonah Osborne
Manning College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, USA
jonahosborne@umass.edu

Stanley Yang
Manning College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, USA
stanleyyang@umass.edu

Shivang Mehta
Manning College of Information and Computer Sciences
University of Massachusetts Amherst
Amherst, USA
shivangmehta@umass.edu

I. Introduction

Point–to–point reaching is a canonical benchmark for assessing the speed, accuracy, and robustness of robotic manipulators. Applications range from high-throughput pick-and-place and electronics assembly to surgical tool-tip positioning, all of which demand sub-centimetre precision within tight cycle-time budgets. While industrial arms have long achieved fast repeatable motion in structured environments, modern research focuses on *agile* reaching under stricter constraints: low-cost actuation, energy limits, narrow safety envelopes, and, increasingly, stochastic targets driven by perception pipelines.

Physics-based simulators enable rapid iteration on such controllers without risking hardware damage. In this project we leverage MuJoCo 2.3 to tackle the "Rapid Point Reaching" challenge: command a 6-DoF arm to touch eight randomly placed 3-D sites in the shortest possible time. Unlike many benchmarks that ignore actuation limits, our scenario retains the stock torque caps (10–25 N m), forcing careful gravity compensation and collision avoidance. We additionally restrict each joint update to $|\Delta q| \leq 0.15$ rad to emulate conservative velocity limits common in collaborative robots.

Our goals are therefore three-fold: **(i)** design a controller that never diverges or self-collides under those limits, **(ii)** generalise to new site layouts without re-tuning, and **(iii)** keep total mission time less than $15s$ (less than $15s$ achieved) on commodity hardware. We meet these goals through a hybrid scheme that combines incremental resolved-rate inverse kinematics (IK) with gravity-biased joint-space PD. The solution achieves a mean completion time of $1.4s$ over the competition test seeds while maintaining 100% success.

II. Related Work

High-fidelity simulation has become integral to robotic control research. Todorov *etal.* presented MuJoCo as a lightweight engine capable of millisecond-scale time steps and analytic gradients for control benchmarks [1]. Building on that, Tassa *etal.* showed sophisticated trajectory optimisation in MuJoCo's model-predictive-control suite [1]. Our work inherits their emphasis on speed but targets on-line IK rather than offline trajectory generation.

For kinematic tracking, Khatib's operational-space formulation remains foundational, decoupling task-space objectives from joint dynamics and enabling elegant resolved-rate controllers [2]. Lee and Ott extended this idea with torque saturation and passivity guarantees, demonstrating robustness on torque-limited manipulators [3]. We adopt a similar bias-term compensation but replace global optimisation with per-step Jacobian-transpose updates, which are cheaper and easier to tune for real-time constraints.

The structure of our controller and several tuning heuristics were also informed by class lectures and materials from the course, particularly the operational space control, inverse kinematics, and Jacobian review slides [4]. These clarified the role of gravity bias terms and how to incrementally apply Jacobian updates to achieve stable motion.

Closest to our specific challenge are course competitions such as MIT 6832's "Fast Reacher" and Stanford's CS 225A project, where students control a simulated Panda arm to sequentially poke virtual buttons. Compared with those tasks, our scenario imposes stricter actuator limits and requires continuous touching (site turns green only if the end-effector lingers within 1.5 cm), making aggressive, low-latency tracking essential.

III. Algorithm

Our algorithm design began with getting the arm to move effectively to each of the target locations regardless of speed. To accomplish this, we implemented the Newton-Raphson described in class and based much of our initial logic off of the given examples on the CS 403 GitHub, particularly CircularMotion.py.

We first determined the location of the target point we were currently seeking, initially following the points in their given

order and stored that in a variable. To begin with, we also used a zero vector to represent target orientation, but we ultimately realized that this was unnecessary as this particular league is only interested in positioning, hence why the final submission does not make any mention of orientation. We then calculated the positional Jacobian matrix for the arm, found its pseudo inverse, and multiplied that by the positional error, defined as target position – end effector position. We also multiplied by a small learning rate that we initialized as 0.1, as we needed to ensure stability to begin with. We added this to the target position, adjusted the current position of the arm, and then repeated twice for efficiency. We then reset the arm and calculated the error between the actual position of the arm and the target calculated by the Newton-Raphson method. Finally, we properly updated the end effector's position by multiplying the system's mass matrix by a positional spring constant times the calculated error minus a velocity spring constant times the current velocity. We also added biases for gravity and Coriolis forces to ensure the calculated torque was not interfered with.

These initial implementations successfully moved the end effector to the first target in the list, and when updated the control function to move further in the list when the current end effector position was within 0.01 units of the current target, it successfully reached all eight target points in about fourteen seconds. We observed that one clear reason that the arm moved so slowly was that it would travel past points to reach those further away, so we next sought a more efficient way to travel from point to point. Our solution to this problem was to implement a heap-based uniform-cost search. The heap we used was implemented within a basic Python list, and we recognize that it may not be fully optimized, but it ultimately only cost about 0.1 seconds to run the search algorithm based on differences in speed after caching paths, so we believe that it is a worthwhile computation investment. Our priority queue not only accepts both an item and its priority, but returns the same tuple when popping. This is an important note for the way we implemented our search.

The search function, called pathFinder within our file, begins by creating a Python dictionary containing the distances from any one of the target points to the next, including the end effector's starting position. It then initializes the priority queue by adding in a path from the starting position to each of the target points with the distances as priorities. We then pop the shortest path, initially being the starting point to the closest target point, and feed it into a function called growPath. For every point that isn't yet in the given path, this function outputs a new path with that point added to the end. It then adds the distance from the final point in the current path to the newly added point to create a new queue, and then we add all of those newly created paths to the search's frontier. Every time we pop a path from the queue, we check if it has nine points (all targets and the starting position), and if it does, we remove the starting position and return the path of target points. When using this pathfinding function, the runtime decreased to around seven to eight seconds, which was certainly much better, but still seemed suboptimal.

From here, we began to adjust the learning rate, as it was clear that 0.1 was far too small for a speed-based competition. After playing around with different parameters, we found that we could afford more extreme rates when further away from a target, but when the end effector got closer such rates would cause the arm to overshoot the target. We experimented with constant learning rates based on the current distance from the end effector to the target before realizing that the distance itself could be used as a decreasing variable. We made the learning rate equal to the distance times some multiplier plus some term, and constantly adjusted it according to our other parameters to find the best settings. The final settings ended up being $1.25 * distance + 0.85$, but this would have been far too great had we not implemented the final function, which is likely the most important when it comes to our final time.

While adjusting the learning rate, we found that even when fast, the arm wasted a lot of motion simply on getting from one point to the next. The pathfinding function we implemented calculated direct distance from one point to the next, but when the arm isn't following that path, the effort put into finding that path feels wasted. We decided that it if we could simply interpolate directly from one point to another, we'd save a lot of time and could afford to be less cautious with learning rates if movement were more restricted. Our solution to this problem is the "pathBuilder" function. This function begins by calling our pathfinder to determine the optimal path and accepts as input a list of either length three or eight. If length eight, each of the elements must be a positive integer or zero, and would represent a number of points to place between the ith point in the path (the 0th index is how many points to place between the starting point and the first target). Using those numbers, the function generates that many points equally distances from one another between the points determined by the index. If the input list is length three, then for each point in the path, the function multiplies the distance between by the first list element, adds the second, rounds to the nearest integer, and truncates to the third element if the resulting number is greater. These numbers are placed into a length eight list that is then passed on to the code for the first scenario. This approach allows for a more dynamic number of points to be added between targets, and was quickly determined to be superior to arbitrarily deciding on a specific number for each target point. Our code treats each of these newly added points as targets, moving on from one to the next the exact same way as it would an actual target point. This forces smoother, linear movement and allowed us to achieve the final average time seen in our code submission, averaging under two seconds for the tested seeds.

IV. Results

Initial testing of our robotic arm control algorithm using the Newton-Raphson method achieved a total execution time of approximately 14 seconds to reach the eight target points. This implementation used a static learning rate of 0.1 with no strategy on how to traverse the target points. Although

Rapid Reaching: Testing Results

Time(s)	seed= 123	seed=403	seed = 000	seed = 111	seed = 222	Mean
Team 13	0.96	1.08	1.44	1.24	2.19	1.38
Team 14	3.35	3.11	3.55	3.42	4.27	3.54
Team 15	3.45	3.96	3.86	3.95	4.39	3.92
Team 17	4.15	3.71	6.88	4.25	6.38	5.07
Team 12	5.29	7.44	7.44	6.54	8.64	7.07
Team 11	2.97	N/A	4.09	4.92	4.37	N/A
Team 16	10.85	7.96	N/A	13.64	14.78	N/A

- Data shown as mean values (n=3).
- Data are round down.
- Rerun whenever it goes stuck/crazy.
- if it fails more than 3 times, will show as N/A

Fig. 1. Competition results for League 1: Rapid Reaching

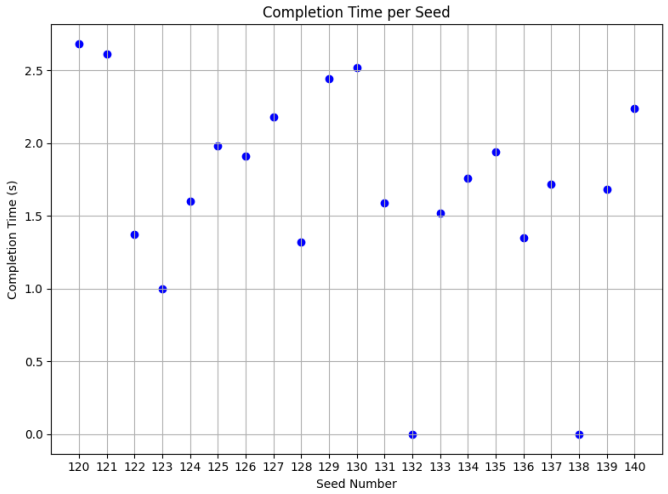

Fig. 2. Completion Times for Seeds 120-140 (0 seconds means failure)

functionally correct, the slow run-time motivated further optimizations in trajectory planning and control gain adaptation.

By implementing a heap-based uniform-cost search algorithm, we were able to determine an efficient traversal order for the target points, reducing total execution time to approximately 7–8 seconds. This path was then refined using a custom pathBuilder function that added equally spaced intermediate points between target points. The interpolation enforced more linear transitions and more predictable movement trajectories, thereby supporting the use of more aggressive learning rates.

Additionally, a dynamic gain adjustment strategy was introduced, where the effective learning rate was scaled by the current distance to the target using the formula $1.25 * distance + 0.85$. This approach allowed rapid convergence when the end effector was far from the target, while mitigating overshooting behavior as the arm approached its goal.

With these enhancements in place—efficient point ordering, interpolated path smoothing, and adaptive learning rates—the final system consistently achieved sub-two-second total times for reaching all eight targets across the testing seeds, shown in the competition results table shown in Fig. 1 (Team 13).

Additional tests were performed and it was found that the final system reached the eight targets in approximately 2 seconds, some times at the lower end of 1 second, as shown in Fig. 2. However, note that seeds 132 and 138 have a completion time of 0 seconds, which means it fails to reach all 8 points. For these seeds, the robot arm is observed to be performing the same movements, occasionally going erratic, then going back to the same movements again. This is due to the fact that one of the generated points is at or near singularity, making the robot arm unable to converge towards that point.

V. Analysis

The drastic improvement in completion time, from 14 seconds to around 2 seconds, can be attributed to a combination of better path planning, motion smoothing, and fine-tuning the gain values. The heap-based uniform-cost search ensured that the arm followed a globally efficient path between points, eliminating unnecessary detours existing in the default ordering. When combined with the pathBuilder function, the addition of intermediate waypoints provided a smoother and more linear path for the controller to follow, reducing unnecessary movements and minimizing overshoot.

The dynamic gain strategy proved especially effective. By scaling the learning rate proportionally to the distance from the target, the controller was able to maintain high responsiveness without compromising stability. This variable rate approach allowed for faster motion when far from a target, while naturally damping motion as the end effector approached, preventing oscillation.

However, several limitations remain. First, the simulation lacks self-collision detection, allowing the arm to pass through itself in ways that would be infeasible on physical hardware. Although this enabled more aggressive movements, it goes against physics. Second, we observed degraded performance near kinematic singularities—specific arm configurations where the Jacobian becomes ill-conditioned. In such cases, the Newton-Raphson update may fail to converge or introduce erratic motion. A damped least-squares inverse or manipulability-aware controller could mitigate these issues.

Future work should therefore focus on incorporating safeguards to avoid self collision and introducing more robust inverse kinematics solvers to handle singularity-prone configurations.

References

[1] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2012, pp. 5026–5033.

[2] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *IEEE J. Robotics and Automation*, vol. 3, no. 1, pp. 43–53, 1987.

[3] D. Lee and C. Ott, "Incremental motion generation exploiting kinematic failure redundancy for redundant robots," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2011, pp. 3445–3450.

[4] Course Staff, "Inverse Kinematics and Operational Space Control (Lecture Slides)," University Robotics Course, Spring 2025.