Dart-Throwing on the KUKA iiwa

Valerie Chen
vkchen@mit.edu

Gregory Xie
gregoryx@mit.edu

***Abstract*—We present an implementation of dart-throwing behavior on a simulated Kuka iiwa robot in Drake [1]. We formulate an optimization problem to find release configurations and explore various methods to move the iiwa to and away from the release configuration. Rudimentary dart aerodynamics are modelled in for more realistic simulations. Our system is capable of throwing darts within an accuracy of 9 cm of the dartboard center at a distance of two meters. However, there are various scenarios under which our system fails at throwing a dart that hits the dartboard; we analyze several of the complexities of dart throwing with a robot arm, such as gripper release dynamics, constraints on dart motion, and modelling in simulation.**

I. Introduction

The game of darts is a widely-played one, and in a range of situations: at bars, in arcades, and at professional competitions. The game consists of two players throwing darts, sharp projectiles with fins, at a round dartboard with their bare hands. Points are scored by hitting specific sections of the board [2].

The act of throwing the dart is a complex one, and has been analyzed in clinical settings. In [3], uncontrolled manifold analysis was used to study the movement of the different joints in the body when subjects were asked to throw a dart. One result of this analysis was that there was less variance in the fingers as opposed to the wrist, arm and shoulder. The task of throwing darts is complex enough that it has also been used as a case study to evaluate goniometric (the measure of human joint angle ranges) measurement of wrist motions for clinical assessment and rehabilitation [4].

Inspired by the act of dart-throwing by humans, in this project, we have implemented the ability for a KUKA iiwa robot arm to throw darts in simulation. We assume that the dart starts at a repeatable position in the gripper, and that the location of the dartboard is known. Our main project components are:

1) An inverse dart dynamics controller which takes in the location of the dartboard and outputs a desired gripper pose and velocity;
2) A motion planning system which takes in the desired gripper pose and velocity at the target dart release pose and outputs a joint space trajectory for the robot arm; and
3) Rudimentary dart aerodynamics.

The act of throwing the dart requires modelling the kinematics of the dart flight path, which is actually a more complex task than the modelling of throwing an object such as a ball because of the additional aerodynamic forces on the dart (specifically, due to the dart fins). These forces act to stabilize the dart such that its heading is aligned with its velocity, and impose constraints that the dart cannot rotate along axes perpendicular to the dart body. Thus, for a robot to throw a dart, the complexity of the dart-throwing behavior requires additional constraints on the pose and velocities of the robot manipulator as it executes the throw.

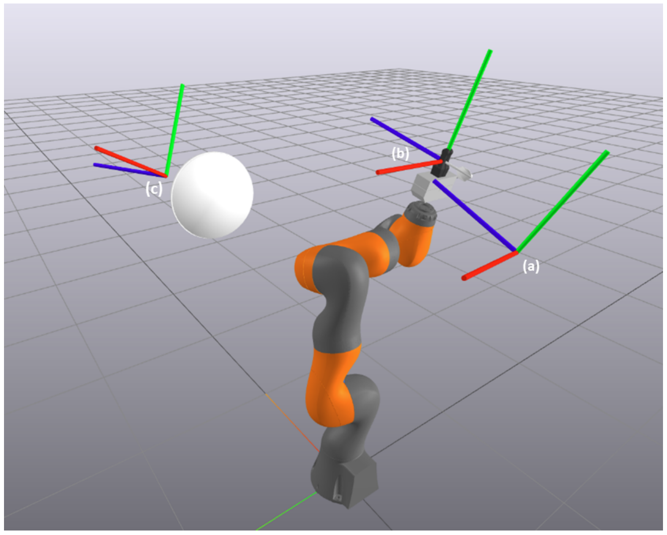

Fig. 1. System setup in simulation. A KUKA iiwa holding a dart in its parallel-jaw gripper, and a dartboard located 2 meters away from the robot, offset in the y-direction by -.3 meters, and 1.5 meters above the ground. (a) shows the starting pose of the calculated throw trajectory, (b) shows the pose of the release point where the dart separates from the gripper, and (c) shows the ending pose of the trajectory.

II. Related Work

In robotics, dart-throwing has been explored in the context of reinforcement learning. In [5], the authors use a reinforcement learning approach to map from task circumstances to meta-parameters, and demonstrate the success of the algorithm on the task of dart-throwing. A robot trainer that can be used in physical therapy applications is developed in [6] using a model-free reinforcement learning approach, and the system is again evaluated on a dart-throwing task.

In previous 6.843 Robotics Manipulation classes, other students have explored throwing objects with the KUKA iiwa as well [7]. However, as mentioned in the previous section, the geometry of darts introduces complexities that can be ignored for tasks of throwing round objects. Our work explores these complexities, proposes approaches to address them, and reveals further areas of study.

III. Methods

A. Simulation

We set up our simulation in Drake, with a 7-DOF Kuka iiwa arm and a Schunk WSG 50 two-fingered parallel-jaw gripper. The simulation also includes a dart and a dartboard, which we modeled in SolidWorks and converted to URDFs.

Collision models are modelled for the robot arm, gripper, dart, and dartboard. The default kBox collision geometries for the robot arm and gripper are used, and simplified collision geometries are added for the dart and dartboard. Both are modelled as cylinders; the dart fletchings are not represented in the collision geometry, which can cause unrealistic simulations (eg. the dart fletchings passing through the gripper fingers), but was sufficient for our proof-of-concept and analysis.

We visualized our simulation using Meshcat.

B. Dart Aerodynamics

To accurately simulate the flight of the dart, we must simulate aerodynamic forces. For a dart thrown normally with the center of mass before the center of pressure, these forces tend to align the orientation of the dart with its velocity [8].

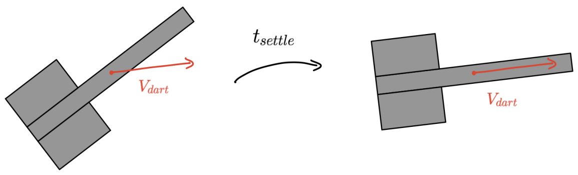

Fig. 2. Example of aerodynamic forces aligning the orientation of a dart with its velocity. The image on the left shows a dart after it has been thrown and before aerodynamic forces have acted on it wit velocity V_{dart}; the image on the right shows a dart after it has been acted on by aerodynamic forces and has settled into that dynamic system after t_{settle} and with velocity V_{dart}. The forces exert a torque on the dart around its center of mass, changing the orientation of the dart.

This is a dynamic system, as these forces and torques act on a body with mass and inertia. To minimize the amount of time that it takes for the orientation of the dart to settle to be aligned with its velocity, the initial orientation of the dart at release should be made to be as closely aligned with its velocity as possible, with minimal angular velocity at release. (Angular velocity around the axis of the dart is acceptable and can actually help to stabilize its flight. However, angular velocity around either of the two axes perpendicular to the body axis should be minimized for our task of dart-throwing.)

These forces grow with the square of velocity, and depend on the surface area of the object perpendicular to the velocity and a drag coefficient. We encapsulate the effects of surface area and drag coefficient into a drag matrix A_{drag}. By multiplying this matrix with the signed square of dart's velocity vector expressed in the dart frame, we can find the aerodynamic forces and torques on the dart. The forces calculated should be applied at the center of mass, as the torques from drag are accounted for separately.

$$f_D^D = A_{drag} sgn({}^W v_D^D)({}^W v_D^D)^2 \tag{1}$$

f_D^D is the vector of torques and forces on the dart expressed in the dart frame and ${}^W v_D^D$ is the spatial velocity of the dart relative to the world frame expressed in the dart frame.

We simulate these aerodynamic effects by adding a new subsystem into our Drake system. This system has an input port for the dart state, and an output port for `ExternallyAppliedSpacialForces`. This subsystem allows us to apply forces on the dart depending on the dart state. From the dart state, we find the rotation of the world frame relative to the dart frame, and use it to find the dart velocity expressed in its own frame. We then use equation 1 to find the forces and torques on the dart in the dart frame, and transform them back to the world frame to be applied.

C. Overall Problem Formulation

To solve for throwing a dart, we formulated the task as two separate problems: finding a release state (pose and velocities) for the robot gripper and generating a trajectory to bring the gripper to and from the release state. We first solve for the release state of the robot gripper, then use the solution release state as an input to solve for the robot arm throw trajectory.

D. Finding Release States

To find the release state of the robot gripper, the flight of the dart to the dartboard must first be modeled. We created a function that took a gripper release pose and velocity as inputs, and returned the time to impact and the ending pose of the dart once it hits the dartboard, assuming a "dartboard" that is an infinite plane at $y = 2$ meters such that the problem can be solved for any gripper pose. For this calculation, we adopted the assumption that the dart acts as a sphere and ignored all aerodynamic effects. This approximation allows for calculation of the ballistic trajectory of the dart while ignoring the complex equations that affect a thrown dart in the real world [8]. As we employ an optimization approach to solve for the gripper release state, we wrap this function so that it returns a squared distance error from the center of the dartboard as well as the time, and takes the joint space configuration of the robot arm and end effector velocity as inputs (`dart_flight`).

We formulate the problem of finding release states as a `MathematicalProgram` optimization problem in configuration space, where our decision variables are our joint angles and velocities. Although many of our constraints are in task space, we choose to do the optimization in configuration space to more easily incorporate joint velocity limits. This is because the mapping from configuration to task space is well-defined while the inverse is not. If we were to do the optimization in task space, the cost and constraint gradients may be less smooth due to the non-unique mapping of end effector state to joint angles and velocities.

Formally, the optimization problem is written as:

$$\begin{aligned} \min_{q,v} \quad & |v|^2 \\ \text{s.t.} \quad & |q| \le q_{max}, \quad |v| \le v_{max} \\ & |J_r^G(q)v| \le \omega_{max} \\ & \text{dot}(f_{kin}(q)_x, J_t^G(q)v) = 0 \\ & \text{dot}(f_{kin}(q)_z, J_t^G(q)v) = 0 \\ & \text{dot}(f_{kin}(q)_y, J_t^G(q)v) > 0 \\ & \text{bb}_{min} < f_{kin}(q) < \text{bb}_{max} \\ & err \le err_{max}, \quad t > 0 \end{aligned} \tag{2}$$

where v is the vector of joint velocities, q is the vector of joint angles, v_{max} and q_{max} are the maximum allowed joint velocities and angles, $J_r^G(q)$ is the jacobian matrix of the robot gripper given joint angles q, ω_{max} is the maximum angular velocity of the gripper allowed, bb_{min} and bb_{max} are the bounding box limits of the release pose, and (t, err) = `dart_flight(q, v)`. $f_{kin}(q)_x$, $f_{kin}(q)_y$, and $f_{kin}(q)_z$ are the velocities of the end effector in the world frame x, y, and z axes, respectively. In our final solution, we set $q_{max} = [2.86706, 1.9944, 2.86706, 1.9944, 2.86706, 1.9944, 2.95433]$ $v_{max} = 5$ rad/s for all joints, $\omega_{max} = 0.5$, $bb_{min} = [0.125, -1.5, 0.4]$, and $bb_{max} = [0.5, 0, 0.875]$. err was calculated as the euclidean distance between the dart position at contact with the infinite dartboard.

The $q \le q_{max}$ constraint prevents solutions from being outside the iiwa joint limits, where they will always result in a self collision. The $v \le v_{max}$ constraint ensures solutions respect joint velocity limits. The $f_{kin}(q)_x$, $f_{kin}(q)_y$, $f_{kin}(q)_z$ and the $J_r^G(q)v \le \omega_{max}$ constraints align the translational velocity of the gripper parallel to its fingers, ensure that the dart is facing in the correct direction, and limit the angular velocity of the dart at release. These constraints are critical to the effectiveness of the optimization solution, as together they determine the amount of time that the dart is not in line with its velocity. This is because these set the initial conditions of the dart angle's transient response, as mentioned in section III-B. We choose to have only the joint velocity in the cost to avoid having to tune cost coefficients.

By changing the bounding box constraint on the task space pose, we are able to intuitively influence the solutions we get. As we expand the bounding box and encompass more of the robot's workspace, we get solutions with the arm more outstretched. This is because we can achieve the same task space velocities with smaller joint velocities by increasing the radius of the throw. However, this leads to complications with leading into and out of the release state, as the lead-in and lead-out trajectories can easily exit the robot workspace.

As we contract the bounding box, we find solutions with the arm coiled up. This makes it easier to plan trajectories in task space, as the solution is well within the robot workspace. However, we have found that this increases the chance of self-collisions drastically.

We find it pertinent to mention that overall, the maximum joint velocity of the solution gripper state was quite high, around 3-5 rad/s. This is due to the constraint on the angular velocity of the gripper; the smaller ω_{max} is set to be, the greater the joint velocities are to maintain a more linear path.

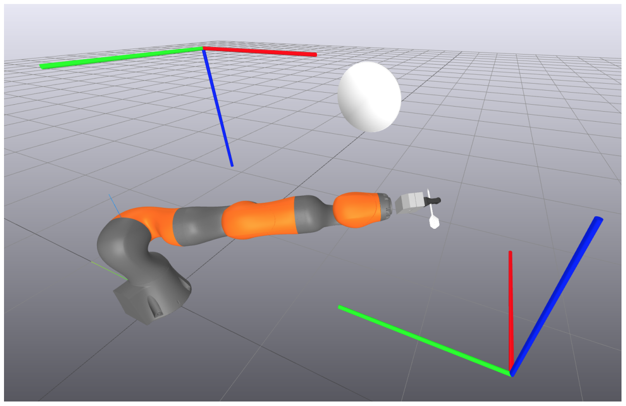
Fig. 3. Example of release state where the arm is outstretched. The large frames visualize the bounding box optimization constraint. The bounding box extends outside the workspace of the robot, allowing the arm to be completely extended. This results in smaller joint velocities, since the dart is at a larger radius.

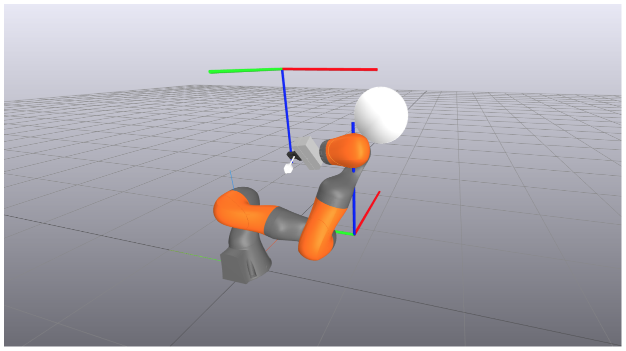
Fig. 4. Example of release state where the arm is coiled. The large frames visualize the bounding box optimization constraint. The bounding box excludes large amounts of the robot workspace, resulting in convoluted arm geometry for the solution release state. This increases the change of self-collisions when naively planning in task space, and requires higher joint velocities than the outstretched case.

Because we have nonlinear constraints, our optimization landscape has many local minima and our solver can get stuck, returning poor solutions. We run the optimization repeatedly with different initial guesses and keep the best solution to increase our chance of finding the a desirable local minimum.

E. Release Trajectory Generation

To throw the dart, we need to release the dart at the release position with the release velocity. We must generate a trajectory during which we can ramp up to and down from the release velocity.

1) Task-Space Linear Trajectory: We first tried a naive approach to generate trajectories that constantly accelerated to and decelerated from the release velocity through the release pose, along the direction of the release velocity.

Using forward kinematics and the gripper Jacobian, we find the pose and velocity in task space. We specify a target ramp-up and ramp-down time, and then use the velocity along with these times to determine the start and end poses from the release pose. For the ramp up motion, we use the full velocity to find the start pose. However for the ramp down motion, we only use the translational component. This is because if the rotational component was also used, the gripper could rotate and hit the dart after release.

We then calculate the required acceleration, and create a time series of poses. In 1D, the equations to find the poses and accelerations are:

$$p_{start} + \frac{a_{up}t_{up}^2}{2} = p_{release}, \quad v_{release} = a_{up}t_{up} \tag{3}$$

$$p_{release} + v_{release}t_{down} + \frac{a_{down}t_{down}^2}{2} = p_{end},$$
$$v_{release} + a_{down}t_{down} = 0 \tag{4}$$

While simple, this approach left a lot to be desired. When applied to the solutions in which the arm was outstretched, the generated trajectories frequently exited the robot's workspace. When applied to solutions with the arm coiled, the generated trajectories frequently caused self-collisions.

2) *Configuration-Space RRT*:* To generate a trajectory that avoids self-collisions and respects workspace limits, we tried a sampling based path planner, choosing to implement RRT* over vanilla RRT because it is asymptotically optimal, which results in "smoother" paths [9]. As these trajectories are used to ramp up to and ramp down from the release velocity with the robot arm, smoother paths are desirable because they increase the chances of the dart reaching the desired velocity at throw with complex gripper-dart dynamics, and they are also safer for a physical robot to execute. We sample points in configuration space, and use forward kinematics to check for collisions in task space.

We picked start and end configurations ($[0, 0, 0, 0, 0, 0, 0]$) that we predicted would have fewest obstacles on the way to our release configuration. Using RRT* to generate paths, we attempted to create constant-acceleration trajectories by finding the total length of the path, and using the Euclidian norm of the release velocity to find the Euclidian norm of the acceleration along the path. We planned paths to and from small perturbed configurations along the direction of the release velocity from the release configuration to better allow the gripper to reach the desired release velocity. We then created a trajectory from the path.

While we were able to generate trajectories which respected workspace limits and avoided self-collisions, in order for these trajectories to be relatively "smooth" without any post-processing, the steering distance had to be small. When finding a path between two points in configuration space that are far away, the small steering distance causes the RRT* tree to grow very large. This resulted in issues in Deepnote where we would run out of memory and the Jupyter kernel would crash. In addition, the trajectories generated would sometimes make the gripper turn quickly right before the release, as the velocity of the approach would often be in the wrong direction.

Because we only need to avoid self-collisions, the configuration space is relatively obstacle-free. We could have fixed these memory issues with more goal sampling, which would have biased the tree towards the goal more, finding a solution faster. Once we realized that the configuration space was relatively obstacle-free, we realized that we could switch to a much simpler of method of generating trajectories.

3) *Configuration-Space Linear Trajectory:* Finally, we generated trajectories by applying the method described in section III-E1 in configuration space. We find the endpoints of the line by stepping forward and backward along the direction of the velocity vector, and stopping the first time we encounter joint limits or a self-collision. Using the configuration space velocity and the endpoints, we are able to find the acceleration in configuration space and generate a trajectory.

This method has one benefit over doing a linear trajectory in task space. For release poses near workspace limits, this method will find a valid trajectory with a reasonable length, while a making linear trajectory in task space will find a very short trajectory if workspace limits are considered.

Compared to RRT*, this method is much faster and deterministic. In addition, it does not have any of the sudden turns before release that we observed in our RRT* trajectories.

Around the time of dart release, the gripper moves in approximately the same direction of the velocity in task space, creating a short follow-through portion of trajectory that helps avoid dart collisions with the gripper. If we linearize the gripper pose and joint configuration relationship around the release pose (q_r), according to the following:

$$X^G = f_{kin}^G(q_r) \tag{5}$$
$$f_{kin}^G(q_r + \Delta q) \approx f_{kin}^G(q_r) + J^G(q_r)(\Delta q) \tag{6}$$
$$f_{kin}^G(q_r + \Delta q) \approx f_{kin}^G(q_r) + J^G(q_r)(v\Delta t) \tag{7}$$
$$f_{kin}^G(q_r + \Delta q) \approx f_{kin}^G(q_r) + \Delta t V^G \tag{8}$$

we see that the change in gripper pose is linearly related to the change in joint configuration for small changes in joint configuration. The change in joint configuration can be written as the joint velocity multiplied by a small change in time. The change in time is a scalar, and we can see that for times close to the release time (small Δt), the gripper moves in approximately the same direction of the task space release velocity. For some release states, this small follow through motion is enough. For others, the gripper still collides with the dart.

In the final dart throwing simulations, we used trajectories created using this method.

F. Results and Evaluation

To analyze the performance of our dart-throwing system, we tracked the distance between the dart frame, located near the center of the body of the dart, and the dartboard frame, located at the center of the dartboard. Ideally, we would track the distance of the tip of the dart at the moment of contact with the dartboard and take the 2-D distance in the Y-Z plane between the tip of the dart and the center of the dartboard, as in a real game of darts. Our method of taking the smallest distance is also inaccurate in the case where the dart hits the board above the center and drops down due to gravity; however, given that we are aiming for the center of the board, our calculations are in the ideal case where the dart reaches the target velocity of the gripper, and that we model drag forces, our dart does not get thrown above the center of the dartboard. Our basic metric is a viable heuristic for preliminary evaluation of our system.

Using the metric described above, our best throw was within 9 cm of the dartboard center. For this throw, we constrained the release state optimization problem to find solutions that were within 5 cm of the dartboard center. To evaluate how well the simulated dart path matched the predicted path used in optimization for our best throw at each timestep while the dart is in flight, we calculate the distance between the predicted location of the dart and the position of the dart in simulation for our best throw, shown in Figure 5.

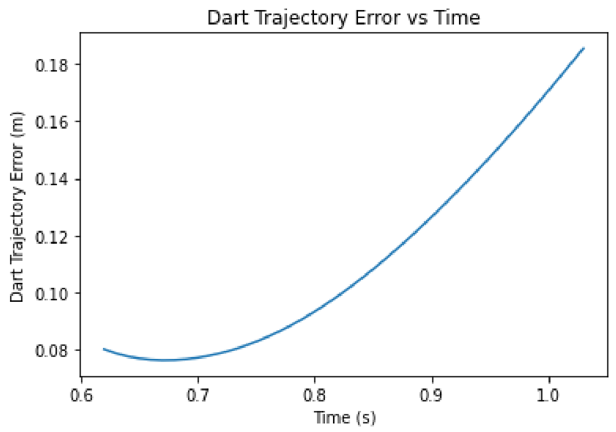

Fig. 5. Error (Euclidean distance) between the predicted and simulated dart trajectory after release for our best throw. The error follows a parabolic trend, which is expected since both trajectories are approximately parabolic. A cumulative time lag between the two trajectories is likely a cause of the large final error, which is larger than the difference in dartboard contact locations.

We see that the error looks parabolic. This is expected, as both the predicted and simulated dart path should be roughly parabolic, and so the distance between the two should also be parabolic. The maximum error between the trajectories was larger than the error between the predicted and simulated dartboard contact locations. This may be because the drag forces slow the simulated dart down, causing it to lag behind the predicted dart, and because the dart does not reach the desired velocity at release.

Additionally, the dart is not released with its orientation in line with its velocity. This is because the dart slips rotationally in the gripper during the ramp up portion of the throw trajectory, and because of the simulation is started.

The rotational slippage while the dart is in contact with the gripper may be because of the contact modeling between the gripper and the dart. Because they are both rigid bodies, the exact contact points are ambiguous. It is possible that the contact points are close enough together that the frictional forces from the gripper are not large enough to react the inertial torques from rotating the dart, causing some slippage. In addition, the dart is cylindrical and the fingers are flat; the contact area between the two is relatively small to begin with.

We start with the gripper fingers close to, but not touching the dart. This is to avoid starting the simulation with the dart and gripper fingers in penetration with each other. At the start of the simulation, we close the gripper. However, there is a small period of time during which the dart can fall and rotate freely. This also contributes to the initial rotational error between the dart and gripper frames.

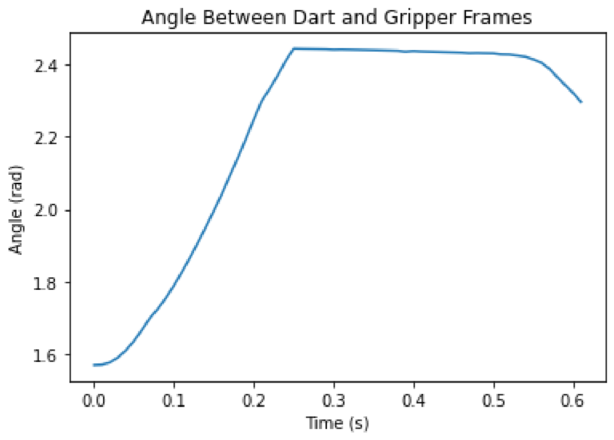

Fig. 6. The angle (error) between the dart frame and the gripper frame during the ramp portion of the throw trajectory for our best throw. There is an initial error from the dart being free to rotate before the gripper can come into contact. Inertial torques during the ramp portion cause the error to increase until the dart hits the base of the gripper and stops rotating. Near the release time, the arm swings the other direction and inertial forces cause the error to decrease slightly before release.

Because the larger side surface area of the dart is more perpendicular to the dart velocity for a longer time, the dart experiences larger drag forces than if it was released straight, slowing it down and making the dart hit the dartboard at a lower position than predicted.

We see the effects of our simulation start state in the distance between the dart and expected dart position during the ramp up portion of throw trajectory. The dart is allowed to drop at the beginning of the simulation, so there is an initial error. During the throw trajectory, the dart frame actually moves closer to the expected position, releasing with a translational error of around 2 cm.

Our method produces throws highly dependent upon the solution found by the optimizer for the gripper release state, commonly producing poor throwing behavior. The two most common causes of failure are the dart being thrown too low and collisions between the gripper and the dart after release. A YouTube video of a few sample throws is linked at the end of this section. The only parameter that was changed between throws was the bounding box to show a variety of release states and trajectories.

We believe that the magnitude of the aerodynamic effects are too large, which significantly changes the dart trajectory.

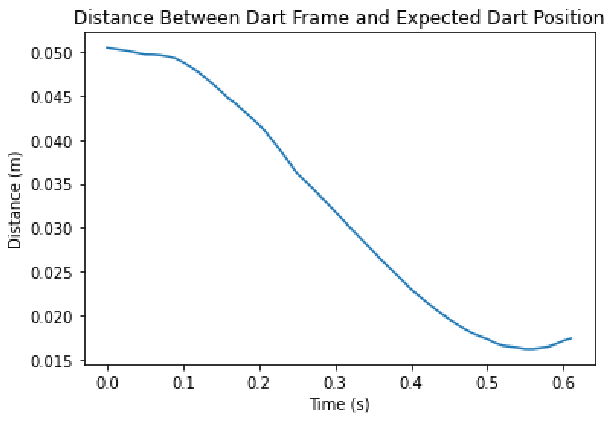

Fig. 7. The translational distance (error) from the dart frame to the expected position of the dart frame in the gripper during the ramp portion of the throw trajectory for our best throw. There is an initial error from the dart dropping slightly before the gripper can come into contact, but forces during the ramp up portion of the throw trajectory happen to decrease the error.

This would continuously decelerate the dart, reducing the range of the throw and making the dart land short of the target. This can be seen from Figure 5, as even for our best throw, there is a significant difference between our simulated dart path and predicted dart path. We estimated the terms in the drag matrix using the drag coefficients of common shapes and the frontal surface areas of the dart along the dart frame axes, which has the potential for large errors. For throws where the projected dartboard contact location is above the center of the board, this results in the dart still hitting the board, but for throws where the predicted contact location is below the center, the dart misses the board.

Our method for generating the throw trajectories occasionally causes the gripper to collide with the dart after release. This happens when the release configuration is in the middle of the configuration space. Since our method has no limit of how far the end configuration can be from the release configuration, the arm swings until it reaches a joint limit or the arm collides with itself. Because the path the gripper takes in task space is not a line, the gripper hits the dart in the process. We could prevent this by adding a distance limit between the end and release configuration.

Video of successful and failed throws: https://youtu.be/PHKEdM61tIw

IV. Conclusions and Future Work

Overall, our implementation of a robot dart-thrower produces viable solutions that throw the dart within 9 cm of the target, but it is not very robust. The follow-through portion of the throw trajectory greatly effects the accuracy of the throw, as the gripper sometimes collides with the dart after release. The accuracy is highly dependent upon the release state found by the optimization problem.

A basic implementation of aerodynamics helps close the sim-to-real gap, adding an effect that is often unmodeled.

In the future, we would like to explore better metrics for evaluating the effectiveness of our dart-throwing strategy. In addition, the inclusion of more sophisticated methods to generate the throw trajectories would decrease the variance in the accuracy of different release states. Formulating the entire dart-throwing problem as one large optimization problem may lead to interesting robot behaviors. For example, a whip-like throw trajectory which equally utilized as many joints as possible would be interesting to observe.

V. Team Member Contributions

Gregory:

- Implementation:
  - Naive task-space linear trajectory generation, RRT*, configuration-space linear trajectory generation
  - Dart aerodynamics
  - Release state optimization
  - Data collection and visualization
- Writing:
  - Simulation, Dart Aerodynamics, Finding Release States, Release Trajectory Generation, Results and Evaluation
- Video content

Valerie:

- Implementation:
  - Simulation setup
  - Release state optimization
  - RRT*
  - Data collection and visualization
- Writing:
  - Abstract, Introduction, Related Work, Simulation, Finding Release States, Results and Evaluation, Figures
- Video content and creation

References

[1] R. Tedrake and the Drake Development Team, "Drake: Model-based design and verification for robotics," 2019.

[2] "Darts," Dec 2021.

[3] J. Nakagawa, Q. AN, Y. Ishikawa, H. OKA, K. TAKAKUSAKI, H. YAMAKAWA, A. Yamashita, and H. Asama, "Analysis of human motor skill in dart throwing motion at different distance," *SICE Journal of Control, Measurement, and System Integration*, vol. 8, pp. 79–85, 01 2015.

[4] V. Vardakastani, H. Bell, S. Mee, G. Brigstocke, and A. E. Kedgley, "Clinical measurement of the dart throwing motion of the wrist: variability, accuracy and correction," *Journal of Hand Surgery (European Volume)*, vol. 43, no. 7, pp. 723–731, 2018.

[5] J. Kober, E. Oztop, and J. Peters, "Reinforcement learning to adjust robot movements to new situations," *Robotics: Science and Systems VI, 1-8 (2010)*, 01 2011.

[6] C. Obayashi, T. Tamei, and T. Shibata, "Assist-as-needed robotic trainer based on reinforcement learning and its application to dart-throwing," *Neural Networks*, vol. 53, pp. 52–60, 2014.

[7] D. Yang and T. Wang, "Throwing in drake," 2020.

[8] D. James and J. Potts, "Experimental validation of dynamic stability analysis applied to dart flight," *Sports Engineering*, vol. 21, no. 4, p. 347–358, 2018.

[9] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *CoRR*, vol. abs/1005.0416, 2010.