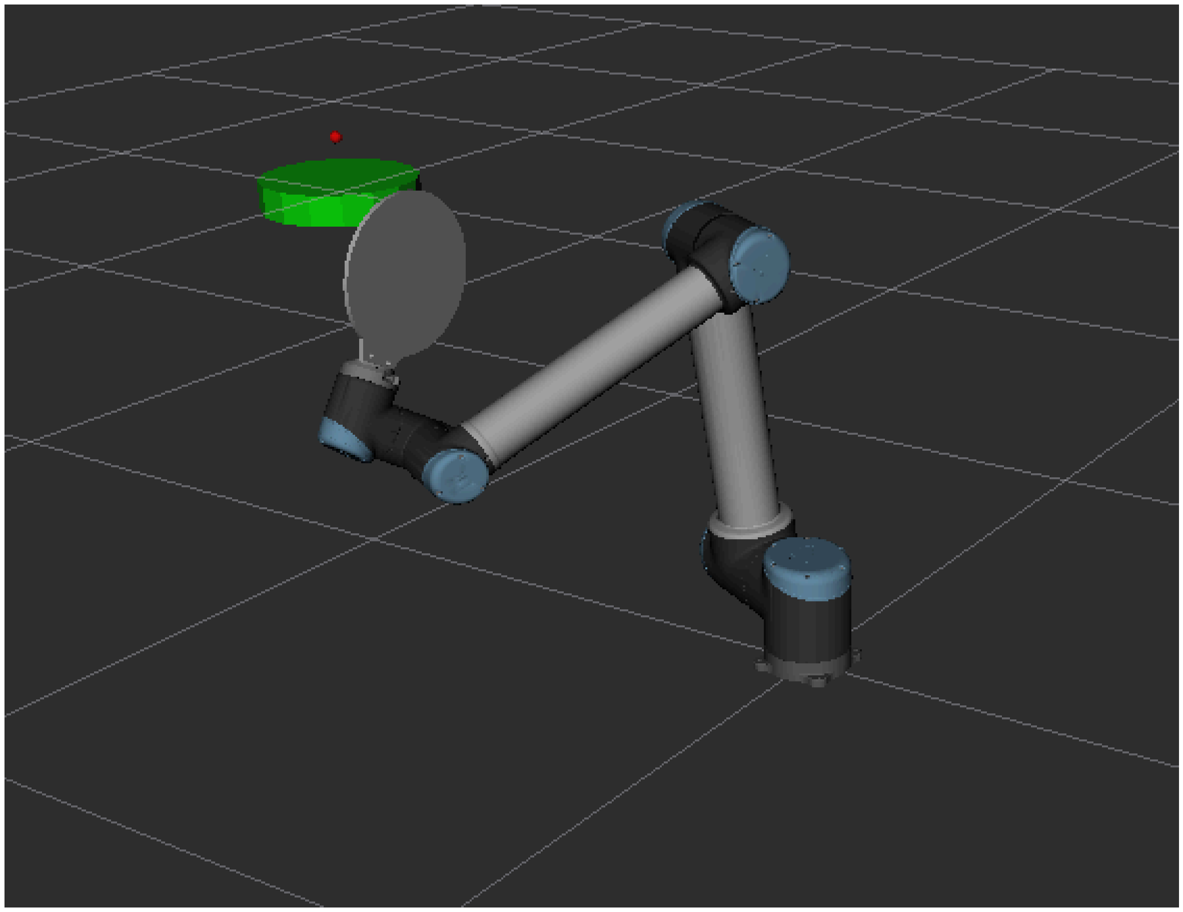
Fly Swatter Robotic Arm

Baaqer Farhat & Andrey Korolev



(Figure 1: Rviz environment including Robot, paddle, ball, and basket.)

Project Description:

This project aimed to control a 6-DOF robot to hit a ball (fly) in 3-D space at a desired target (container) on the ground. To simplify the problem, the fly was assumed to have negligible velocity relative to the robot movement, so it was initialized as a stationary object unaffected by gravity. Once the fly is struck by the robot, it begins to move in the according direction with gravity pulling it downwards.

The robot's base was placed on flat ground at a height of 0m. Therefore, the fly's position could not be physically lower than 0 m or inside the robot's base, which was approximated as a sphere at (0 m, 0 m, 0 m) with a 0.25 m radius. This required that no part of the robot went below a height of 0 m. This constraint influenced one of the major focuses of the robot, which was repelling the robot's trajectory from the ground and base.

This project utilized the UR10 robot arm, with a paddle attached at its tip to hit the fly. The robot was provided only with the fly's position and the basket's position, which it would use to calculate the trajectory required to hit the fly as close to the basket as possible.

Physics Involved:

We opted to use Rviz and our custom physics engine instead of other environments such as Gazebo since the physics we needed to have was simple to custom design. In Rviz, we load the robot URDF and use Visual Markers to represent the fly and basket as a ball and cylinder, respectively. The radius of the ball (sphere) and basket (cylinder) are 0.02 (m) and 0.25 (m) respectively.

1. **Collision of ball with paddle**

The velocity the fly undertakes is determined by the velocity of the paddle at collision. Since the ball represents the fly's position at impact, it remains stationary until hit. At impact, we set the ball's velocity to the paddle's velocity.

2. **Calculating the Ball Trajectory**

When a ball is generated, the goal is to find the optimal initial velocity vector V_0 required for the robot to successfully hit the ball into the basket located at P_{final}. The initial position $P_{initial}$

is also the ball position. The kinematic equation used to calculate the projectile motion in 3D space is as follows:

$$p(t) = P_0 + V_0(t) + \frac{1}{2} g \cdot t^2$$

$$V(t) = V_0 + g \cdot t$$

We took acceleration due to gravity to be -1.0 m/s^2 instead of the standard -9.81 m/s^2 for ease of visualization in Rviz. The main goal here is to minimize $||V_0||$ to ensure that the robot uses the least amount of energy necessary to achieve the desired trajectory. Given the initial position P_0 and target position P_{final}, we want to determine the optimal initial velocity V_0 and time of flight T that minimizes the weighted norm of V_0. From our kinematics equation the P_{final} equation becomes:

$$P_{final} = P_0 + V_0 \cdot T + \frac{1}{2} \cdot g \cdot T^2$$

To find the optimal V_0 that minimizes the weighted norm we get:

$$min_{V_0} ||V_0||_w = \sqrt{w_x(v_{0x}^2) + w_y(v_{0y}^2) + w_z(v_{0z}^2)}$$

$$\text{Subject to } P_{final} = P_0 + V_0 \cdot T + \frac{1}{2} \cdot g \cdot T^2$$

Where w_x , w_y , w_z = Weights for each component of the velocity vector. Rearranging the trajectory equation to solve for V_0 we get:

$$V_0 = \frac{P_{final} - P_0 - \frac{1}{2} \cdot g \cdot T^2}{T}$$

Now Substituting V_0 with the optimization objective:

$$min_T ||V_0||_w = \sqrt{w_x\left(\frac{P_{final} - P_{0x} - \frac{1}{2} \cdot g \cdot T^2}{T}^2\right) + w_y\left(\frac{P_{final} - P_{0y} - \frac{1}{2} \cdot g \cdot T^2}{T}^2\right) + w_z\left(\frac{P_{final} - P_{0z} - \frac{1}{2} \cdot g \cdot T^2}{T}^2\right)}$$

This reduces the problem to a single variable optimization over T.

If we have no weights, we default to equal weights for all velocity components, the "V0_norm_debug" function takes an input of time T, calculates velocity using the rearranged trajectory equation and evaluates the weighted norm $||V_0||_w$. It outputs the weighted norm value to be minimized. For optimization, we start with an initial guess of 1 second. Then we use "scipy.optimize.minimize" to find the T that minimizes $||V_0||_w$. For safety we avoid division by zero and negative time. The function at the end outputs the optimized time and optimal initial velocity vector.

3. **Collision:**

We check if the ball hit the paddle by taking the absolute positional differences along the x, y, and z axes between the ball's center and the paddle's tip. These differences are then compared against a predefined tolerance threshold, which is the sum of the ball's radius and an additional collision tolerance of 0.1. A collision is confirmed if the ball is within the tolerance across all axes. Once a collision is detected, we are then able to transfer the paddle velocity onto the ball's. To check if the ball hit the ground, we monitor the vertical position of the ball along the z-axis and when this value becomes less than the ball's radius, we indicate a collision with the ground. We also set the ball's z position to exactly its radius to ensure the ball rests on the ground without penetrating it. The velocity vector of the ball is also set to zero to stop any movement.

To find the norm that we want to hit the ball at, we divide the velocity by its own magnitude to get its normalized vector, which is the normal of the paddle that the ball should be hit at.

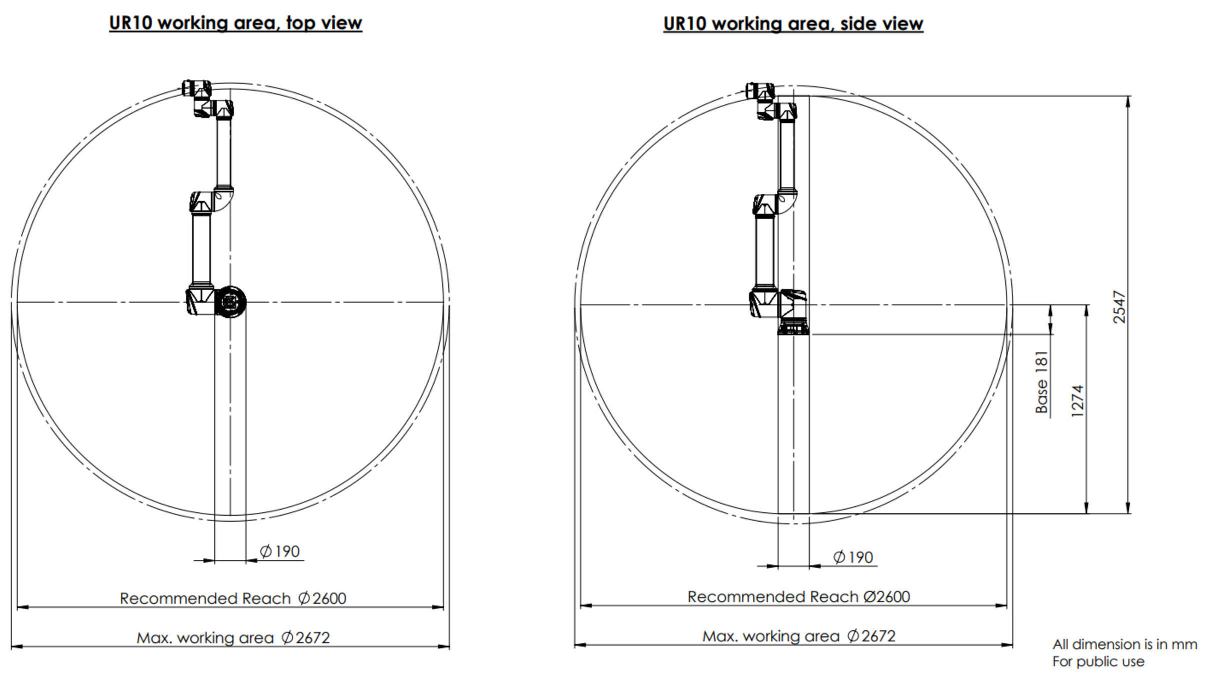
Robot Description:

The UR10 robot we are using has 6 degrees of freedom. We have a fixed base and the following joints:

1. Joint 1 (Base Rotation): Rotates the entire arm around the world z-axis.
2. Joint 2 (Shoulder Lift): Adjusts the vertical position of the arm by lifting or lowering the shoulder, it rotates around the world y-axis.
3. Joint 3 (Elbow Joint): Controls the bending of the elbow, allowing the forearm to move closer to or further from the shoulder.
4. Joint 4 (Wrist 1): Rotates the paddle around the arm's longitudinal axis, enabling the robot to orient the paddle horizontally.
5. Joint 5 (Wrist 2): Adjusts the paddle's pitch, enabling upward or downward tilting.

6. Joint 6 (Wrist 3): Controls the paddle's roll, allowing lateral tilting to achieve the desired orientation.
7. Joint 7 (paddle): Fixed joint attached at the end of Wrist 3.
8. Joint 8 (tip): The tip joint is fixed and is at the center of the paddle.

The robot has a maximum reach radius of 1.3 meters, with an additional 0.25 meters of reach provided by the paddle. The workspace extends vertically and horizontally, allowing the robot to interact with the ball at varying heights and lateral positions. As previously mentioned, the robot is placed on flat ground at a height of 0 m, limiting the potential workspace to only the upper hemisphere of the workspace shown in Figure 2.



(Figure 2: Workspace of the robot)

Task Description:

1. **Primary Task**

The task of hitting a ball to a desired location requires the paddle to be facing a given orientation and moving at a specific velocity. It is possible to fully characterize the robot's desired movement by specifying a tip velocity and tip angular velocity at each instant of operation. While all the vector components must be specified for the tip velocity, the tip angular velocity can be truncated to just two components since we are only concerned with the tilt and roll of the paddle (the pan of the paddle does not affect the resulting direction of the ball). This can be achieved by rotating the jacobian and values of the tip angular velocity from the world frame into the tip frame, where we can simply remove the row corresponding to the paddle's pan axis. Therefore, we have 6 joints and 5 primary tasks: 3 velocity components specified in world frame coordinates and 2 angular velocity components specified in tip frame coordinates. The fact that we have less tasks than joints implies that the system is underspecified and can perform additional secondary tasks.

2. **Secondary Task**

As noted previously, it is desired that our robot does not dig into the ground or its own base during operation. This can occur if the robot reads the ball position to be below ground (case not considered since it can be prevented with a simple check) or, more importantly, if the robot's generated trajectory drives the robot below the ground (this can occur for reasons that will be discussed later). This secondary task was accomplished by applying forces proportional to the elbow's distance from the ground, the tip's distance from the ground, and the tip's distance from the robot's base center. Then, torques were calculated by subtracting the robot base position from the points of interest (elbow, tip, tip) and performing a cross producing with the respective forces. Since only the first three joints (base rotation, shoulder lift, and elbow) can majorly move the robot from the ground, the joint torques were calculated by multiplying the 6x3 jacobian (3 velocity and 3 angular velocity tasks, 3 joints) by the stacked forces and torques. Since all 6 joint torques need to be specified, we assign joint torques of 0 to the remaining 3 joints. As this is a secondary task, it is not guaranteed that it will always be achieved. While this is not favorable in the context of preserving the robot, it can be justified in the sense that hitting the fly is of utmost importance (the fly may be dangerous).

<u>Implementation & Algorithms:</u>

1. **Joint Based Trajectories, Converted to Tasks, Converted back to Joints**

We took a somewhat non-standard approach to implement the desired behavior for the sake of experimentation. During any phase the robot was in (setup, hitting, return), we generated a joint trajectory using a cubic spline given specified conditions (trajectory time, initial joint position, final joint position, initial joint velocity, final joint velocity), which yielded the desired joint positions and velocities. However, instead of feeding those outputs directly to the robot, we performed forward kinematics to retrieve the robot's desired position, rotation, velocity, angular velocity, and paddle normal. These values were then used to calculate the position and rotation errors (which were scaled by lambda=20), performing calculations as though we had position-based trajectories. Then, we performed inverse kinematics given our primary and secondary tasks to retrieve the final desired joint positions and velocities. The reasoning behind this conversion from joint-space to task space back to joint space was to achieve the best of both worlds, whereby we could get the benefit of having a trajectory that simply brought the joints to where they should be while still having that trajectory constrained in the task space.

2. **Calculating Final Joint Hit Position - Newton Raphson:**

To calculate the final joint position used in the cubic spline, we used the newton raphson algorithm. Newton raphson allows one to converge to the final joint position under specified conditions, which essentially encapsulates all our primary task requirements. Since the algorithm can occasionally send itself to an equivalent angle that is wrapped around many times, we unwrap the retrieved guess by 2*pi. The pseudo code is as follows:

```
Make an initial guess for the final joint position
Try 100 times:
        Calculate reference position and rotation with forward kinematics (using
        current joint guess).
        Calculate the reference paddle normal, picking it out from the rotation column
        corresponding to the normal axis.
        Calculate position and paddle normal errors (p_error = pgoal - pr ;
                                                     n_error = nr x ngoal)
        Pick out the rows of n_error corresponding to the tilt and roll axes in the
        paddle frame.
        Stack the errors into one column vector (p_error, n_error_xz).T
        Calculate the adjusted jacobian matrix (explained in next section).
        Calculate the resulting joint positions: psuedoinverse(adjusted jacobian) *
                                                 error_vector
        Add the resulting joint positions to the current joint guess.

Unwrap joint angles using fmod(q, 2*pi).
```

3. Adjusted Jacobian and Adjusted Angular Velocity

For the parts of the robot movement we want to specify using only 2 angular velocities, we must calculate the correct adjusted jacobian and angular velocity. As mentioned previously, we must rotate the angular velocity jacobian and the angular velocity from the world frame into the tip frame:

```
Adjusted Jacobian:

Angular velocity jacobian in tip frame = (current transposed world rotation matrix) *
                                         angular velocity jacobian in world frame
Extract the rows of the angular jac. in the tip frame corresponding to the tilt and
roll paddle axes.
```

```
Stack the velocity jac. in the world frame on top of the truncated angular jac. in
the tip frame.
```

```
Adjusted Angular Velocity (Much like adjusted jacobian):

Multiply current world rotation matrix by the angular velocity vector in world frame.
Extract rows corresponding to the tilt and roll paddle axes.
```

4. Repulsion

The repulsion secondary task was described in fair detail previously. An additional important implementation detail is how we scaled the forces in relation to the distances:

```
elbow_distance_to_ground = elbow height
tip_distance_to_ground = tip height
tip_base_diff = tip position - robot base position
tip_distance_to_base = norm of tip_base_diff
tip_base_diff_directions = vector of signs of tip_base_diff

F_elbow = upwards force 0.1*e**(-elbow_distance_to_ground / 0.02)
F_tip_ground = upwards force 0.1*np.e**(-tip_distance_to_ground / 0.02)
F_tip_base = forces of magnitude 0.1*(tip_distance_to_base / tip_distance_to_base**2)
             with directions corresponding to the negated tip_base_diff_directions
F_total = sum of all forces

Calculate the corresponding torques (explained previously).
Calculate the stacked jacobian (full velocity jac. on top of full angular vel jac.)
Calculate the stack input vector (sum of forces on top of sum or torques).
Joint torques = transpose(stacked jacobian) * stacked input vector.
```

The idea behind the elbow force and tip-ground force being scaled with exponentials is that they look very similar to the classic force calculation - distance / distance**2 - but they also grow larger as the values get negative. Therefore, if the points of interest do cross over z = 0 m, they will be repelled even more. The constants multiplying all three forces scale them, while the 0.02 (equivalent to the decay constant) governs how far from the ground the points of interest should start getting repelled.

5. Inverse Kinematics / Sequences

The robot has three different phases: setup, hit, and return. The setup sequence uses a joint trajectory to bring the robot to one of two "home positions," which is determined by whether the y-position of the ball is positive or negative. While very basic, this setup sequence brings the robot closer to the ball and prevents any unusual trajectories caused by angle wrapping. The return sequence is very similar: it brings the robot from the final joint position to the starting joint position using a joint trajectory. Both of these phases care about all 6 tasks (3 velocity and 3 angular velocity), so they simply use the stacked jacobians and stacked tasks without truncating. The hit sequence, on the other hand, requires the truncated jacobians and tasks as discussed previously. After the full jacobians are created for the respective sequences, all sequences follow the same inverse kinematics:

```
Calculate the svd of the jacobian (U vector, S vec, V transpose vec).
Calculate the modified singular values (1/s_i if s_i >= self.gamma else
                                                     s_i/self.gamma**2)
Inverse jacobian = V * modified S * (U transposed)  # This is targetted
                                                    # removal to prevent
                                                    # reaching singularity.

Secondary joint velocity = result from repulsion
Final joint velocity = (Inverse Jacobian) * (stacked task vector) +
                       (I - (inverse jac. * jac)) * secondary joint vel.
# (I - (inverse jac. * jac)) is the null space of the primary task
```

6. Trajectory Time Calculations

The setup trajectory time is constant - either 0 seconds or 2 second depending on the scenario, since we only have two preset starting positions. The hit time and return time are equal and calculated as follows:

```
Set maximum sequence time to 5 seconds.
Calculate the absolute values of the difference between initial and final
joint positions.
Find the maximum absolute joint difference from above.
Sequence time = 3 * (max joint diff) ** 0.5 / max joint velocity corresponding
to joint with max absolute joint diff.
Cap sequence time at 5 seconds.
```

The notable part is how we calculated the sequence time by multiplying by 3 and taking the square root. These numbers were found by iteratively testing the constants that would yield the sequence times that were of fair length (not too long or short) for a wide range of ball positions. The maximum joint velocities were all arbitrarily set to 2.5 m/s and were not enforced.

7. Miscellaneous Notes

When the fly is outside of the robot's reachable workspace, the robot will still attempt to hit it (most likely not succeeding). Since we consider hitting the fly crucial, the robot still tries to hit it as close as it can. The requires getting as close to the singularity as possible, however we prevent the robot from ever reaching singularity by using a targetted removal, as seen in the inverse kinematics section, with a gamma = 0.23. This value was derived experimentally and was seen to yield the best behavior near singularity while boding well with our ground and base repulsion.

<u>Test Cases Presented in Video:</u>

Start Position:	Target position:	Final Position:	Issues:
(0.5, 0.5, 0.5)	(2, -2, 0)	(2.07, -2.04, 0.02)	None!
(-0.63, -0.63, 0.63)	(2, -2, 0)	(2.07, -2.07, 0.02)	None!
(-1.3, 0.0, 0.3)	(2, -2, 0)	(0.98,-1.4, 0.02)	Successfully Hits the Ball, But does not reach target due to gamma preventing the robot from reaching singularity via slowing it down.
(1.3, 0.0, 0.3)	(2, -2, 0)	(2.06, -2.05, 0.02)	None!
(0.0, 1.2, 0.3)	(2, -2, 0)	(1.05, -0.49, 0.02)	Successfully Hits the Ball but ball doesn't reach basket due to repulsion function
(0.0, -1.3, 0.3)	(2, -2, 0)	(2.12, -2.09, 0.02)	None!
(0.0, 0.0, 1.3)	(2, -2, 0)	(2.04, -2.16, 0.02)	Successfully Hits the Ball and the ball reaches basket

(0.0, 0.0, 1.4)	(2, -2, 0)	(0, 0, 1.4)	Fails to hit the ball since the provided position is out of reach, but it will attempt to hit
(0.6, 0.6, 0.6)	(10, -2, 0)	(9.48, -3.01, 0.02)	Successfully Hits the Ball and the ball reaches very close to basket due to high acceleration and error buildup

For more test cases please check Appendix A at the end of the report.

<u>Conclusions:</u>

While our implementation was undoubtedly non-standard, we gained insight about combining joint trajectories with task-space constraints. The robot's performance was quite successful, however there were indeed instances where the robot's behavior was unpredictable, which could have been due to poorly selected constants and artifacts caused by the secondary task. Future additions to the project could include fully enforcing maximum joint velocities and calculating more accurate trajectories (perhaps with anchor points).

Appendix A:

After the successful completion of normal cases, we composed a series of edge cases that would potentially break the robot and give a deeper understanding of how the robot would react when confronted with extremes. It's important to note however that in cases where $Z < 0.2$, the robot repulsion function does not allow it to hit the ground to avoid the paddle from touching the ground.

Edge Test Cases:			
Start Position:	**Target position:**	**Final Position:**	**Issues:**
(1.3, 0.2, 0.2)	(3, -2, 0)	(2.83, -1.66, 0.02)	None!
(-1.3, 0.2, 0.2)	(3, -2, 0)	(1.50, -0.86, 0.02)	Expected for a point very close to singularity
(0.0, 1.3, 0.2)	(3, -2, 0)	(2.63, -1.48, 0.02)	Expected for a point very close to singularity
(0.0, -1.3, 0.2)	(3, -2, 0)	(2.58, -1.90, 0.02)	Expected for a point very close to singularity
(0.0, 0.0, 1.3)	(3, -2, 0)	(2.26, -1.49, 0.02)	Expected for a point very close to singularity
(0.92, 0.92, 0.2)	(3, -2, 0)	(2.91, -1.72, 0.02)	None!
(0.92, -0.92, 0.0)	(2, -2, 0)	(2.48, -1.58, 0.02)	None!
(-0.92, 0.92, 0.2)	(3, -2, 0)	(0.39, -0.02, 0.02)	Extremely low and near max ball position, expected for an extreme case.
(-0.92, -0.92, 0.2)	(3, -2, 0)	(2.67, -2.01, 0.02)	None!
(0.76, 0.2, 0.76)	(3, -2, 0)	(3.064, -2.03, 0.02)	None!
(0.76, 0.2, -0.76)	(3, -2, 0)	(0.76 0.2 0.02)	Expected, repulsion doesn't allow it to hit ground (-z

			values)
(-0.76, 0.2, 0.76)	(3, -2, 0)	(2.86,-1.92, 0.02)	None!
(-0.76, 0.2, -0.76)	(3, -2, 0)	(-0.76, 0.2 , 0.02)	Expected, repulsion doesn't allow it to hit ground (-z values)
(0.0, 0.76, 0.76)	(3, -2, 0)	(3.07, -2.00, 0.02)	None!
(0.0, 0.76, -0.76)	(3, -2, 0)		Expected, repulsion doesn't allow it to hit ground (-z values)
(0.0, -0.76, 0.76)	(2, -2, 0)	(2.079, 2.09, 0.02)	None!
(0.0, -0.76, -0.76)	(3, -2, 0)	(0.0, -0.76, -0.76)	Expected, repulsion doesn't allow it to hit ground (-z values)
(0.63, 0.63, 0.63)	(3, -2, 0)	(3.03, -1.94, 0.02)	None!
(0.63, 0.63, -0.63)	(3, -2, 0)	(0.63, 0.63, -0.63)	Expected, repulsion doesn't allow it to hit ground (-z values)
(-0.63, -0.63, 0.63)	(3, -2, 0)	(2.84,-1.98, 0.02)	None!
(-0.63, -0.63, -0.63)	(3, -2, 0)	(-0.63,-0.63,-0.63)	Expected, repulsion doesn't allow it to hit ground (-z values)