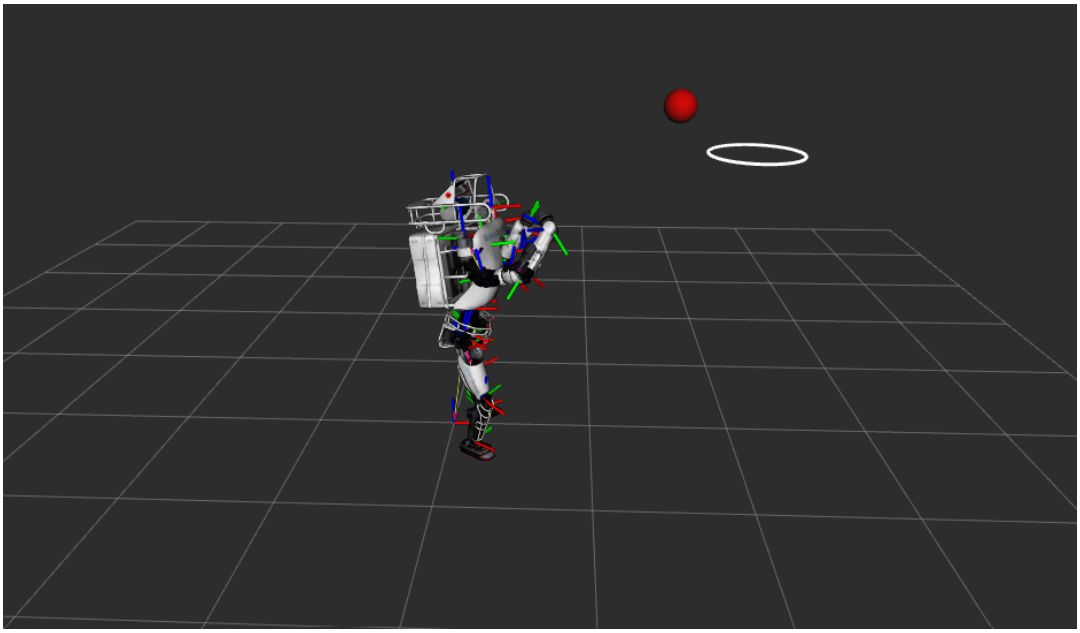
# *The Ballers* Present

## *Rim-Bot!*

By Joao Pedro Bastos, Anya Mischel, and Jabri Garcia-Jimenez

## I. Introduction/Objectives

Our goal of this project was to write an algorithm for ATLAS, a highly advanced, fully-electric humanoid robot by *Boston Dynamics* to throw a basketball into a hoop. We planned to use our understanding of kinematic chains, trajectories, and task prioritization to create code to make ATLAS go from an initial joint position to a final task position of its right hand, where a ball would then follow projectile motion into a hoop with a user-defined location. We used simple equations to control the ball under gravitational acceleration. We set the constraint of both of ATLAS' feet being anchored to the ground to avoid erratic movement of the feet, providing us with stability and allowing freedom in all other joints. We used the provided ATLAS URDF to visualize the robot and referenced the RVIZ demonstrations on Canvas to integrate the visualization of the ball and hoop.



## II. System Description

A. ATLAS

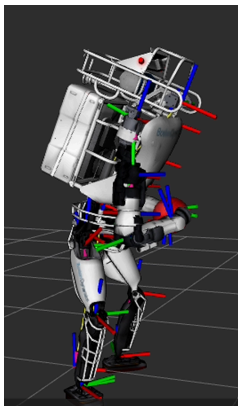
ATLAS traditionally has 30 degrees of freedom. This includes 6 degrees of freedom per leg, 7 per arm, 3 for the back, and one for the neck. The provided URDF for ATLAS made the visualization of the robot much easier since we did not edit it at all.

B. Ball Visualization

In addition to ATLAS, we also integrated a simple 3 DOF ball which accelerates downwards under gravity. We set the position of the ball to the the average of the positions of ATLAS' left and right hands in order to give the appearance that ATLAS was holding the ball between its hands and set the velocity of the ball to be the final velocity of the right hand at the release point. From this velocity and position, we integrated the acceleration each time step to get the updated velocity and integrated the velocity each time step to get the updated position for the duration of time after ATLAS had let go of the ball until it went into the hoop.

C. Tasks and Task Organization

The primary task was to keep the feet planted on the ground so that this could always be guaranteed, no matter the desired trajectories of the rest of the joints to throw the ball. The secondary task was to reach the desired position and velocity of the release point of the ball by the right hand, which the user could specify in XYZ coordinates in the world frame. For most tests of the robot, we chose a right hand release position of [0.3, -0.1, 0.95] and a release velocity of [2.5, 0, 4.5]. We varied the initial joint positions and velocities for different tests, with the main (default) test case being the position of ATLAS displayed below standing upright with slightly bent knees and elbows. The overall size of the big jacobian was 24x30, but broken into individual tasks, the primary task only specified the position of the right foot relative to the left foot, so that section of the jacobian was 6x30, while the section of the jacobian specifying the secondary task was 18x30. We tested for the scenario where the task was outside of the workspace of the robot. When we set the desired position much out of reach of ATLAS, we found that it tends to behave erratically unless we set the damping factor gamma to a very high value, in which case none of the joints move very much, as expected.



## III. Implementation

The algorithm works by creating kinematic chains from the pelvis to each of the appendages, namely to the left foot, right foot, head, and the left and right hands, which both include the joints in the torso. We created a limb class, which takes in the initial joint positions, the kinematic chain, the current joint positions and angular velocities, and the target joint positions. We create a spline from rest to a final hand velocity of the right

hand of [2.5, 0.4, 4.5] in the work space using cartesian coordinates for a duration of 1 second.

```
if t <= 1:
            sp, spdot = spline(
                t,
                1,
                self.right_arm.p0,
                self.right_arm.targets["pright"],
                np.array([0.0, 0, 0.0]),
                np.array([2.5, 0.4, 4.5]),
            )
```

We first perform forward kinematics on the left hand, right foot, and left foot from the pelvis. Using the smaller jacobians computed from these kinematic chains, we filled in a 24x30 jacobian from the pelvis. It should be noted that the Jacobian is only 24x30 since we do not care for the kinematics for the head since they are not relevant to reaching the desired tasks. We eventually wanted to combine the chains so that the right foot and right hand started from the left foot which is defined relative to the world frame, rather than from the pelvis. This way, we could fix the position of the left foot and fix the position of the right foot with respect to the left foot so that both feet remain planted on the ground the entire time. Furthermore, we wanted to link the left hand position to the position of the right hand so that it could always follow it, similar to how a human would when holding a basketball. We used the formulas from the Multiple Kinematic Chains handout to retrieve the Jacobian transformations which we implemented for the new left hand → right foot, left foot → right hand, and right hand → left hand chains as shown in the snippet of code below. This way, we had an 18x30 Jacobian for the right foot and right and left hands.

$$ {}^{\mathrm{f}}\left|J^{v}_{r-f}\right. = {}^{\mathrm{o}}\left|R_f^T\right. \left( {}^{\mathrm{o}}\left|J^{v}_{r-o}\right. - {}^{\mathrm{o}}\left|J^{v}_{f-o}\right. + \left[ {}^{\mathrm{o}}\left|p_{r-o} - {}^{\mathrm{o}}\right|p_{f-o}\right]_{\times} {}^{\mathrm{o}}\left|J^{\omega}_{f}\right. \right) $$

$$ {}^{\mathrm{f}}\left|J^{\omega}_{r}\right. = {}^{\mathrm{o}}\left|R_f^T\right. \left( {}^{\mathrm{o}}\left|J^{\omega}_{r}\right. - {}^{\mathrm{o}}\left|J^{\omega}_{f}\right. \right) $$

```
        self.bigJPelvis = np.zeros((24, 30))
        self.bigJPelvis[0:6, :6] = self.left_leg.J
        self.bigJPelvis[6:12, 6:12] = self.right_leg.J
        self.bigJPelvis[12:18, 12:15] = self.left_arm.J[:, :3]
        self.bigJPelvis[18:24, 12:15] = self.right_arm.J[:, :3]
        self.bigJPelvis[12:18, 15:22] = self.left_arm.J[:, 3:]
        self.bigJPelvis[18:24, 22:29] = self.right_arm.J[:, 3:]

        Jv_lfp = self.bigJPelvis[:3, :]
        Jw_lfp = self.bigJPelvis[3:6, :]
        Jv_rfp = self.bigJPelvis[6:9, :]
        Jw_rfp = self.bigJPelvis[9:12, :]
        Jv_lhp = self.bigJPelvis[12:15, :]
        Jw_lhp = self.bigJPelvis[15:18, :]
        Jv_rhp = self.bigJPelvis[18:21, :]
        Jw_rhp = self.bigJPelvis[21:24, :]
```

```
        p_hf = R_lfp.T @ (p_rhp - p_lfp)
        R_hf = R_lfp.T @ R_rhp
        Jv_lhrh = R_lfp.T @ (Jv_lhp - Jv_rhp + crossmat(p_lhp - p_rhp) @ Jw_rhp)
        Jw_lhrh = R_lfp.T @ (Jw_lhp - Jw_rhp)

        Jv_rhlf = R_lfp.T @ (Jv_rhp - Jv_lfp + crossmat(p_rhp - p_lfp) @ Jw_lfp)
        Jw_rhlf = R_lfp.T @ (Jw_rhp - Jw_lfp)

        Jv_rflf = R_lfp.T @ (Jv_rfp - Jv_lfp + crossmat(p_rfp - p_lfp) @ Jw_lfp)
        Jw_rflf = R_lfp.T @ (Jw_rfp - Jw_lfp)

        self.bigJ = np.zeros((18, 30))
        self.bigJ[0:3, :] = Jv_rflf
        self.bigJ[3:6, :] = Jw_rflf
        self.bigJ[6:9, :] = Jv_lhrh
        self.bigJ[9:12, :] = Jw_lhrh
        self.bigJ[12:15, :] = Jv_rhlf
        self.bigJ[15:18, :] = Jw_rhlf

        Jf = np.vstack(self.bigJ[:6, :])
```

We then extracted the part of the Jacobian for the primary task of keeping the feet planted on the ground and performed a damped and weighted inverse on the big jacobian for the secondary task (which didn't matter for the rows of the jacobian corresponding to the primary task since it is not prioritized). We elected to use a damped and weighted inverse for the jacobian of the secondary task for two main reasons. Having the jacobian for the second task be damped is helpful because it avoids erratic behavior around the singularity. The secondary task was to move the hands to the desired position and velocity, so this naturally caused the arms to run into a singularity when they were outstretched. Having the jacobian also be weighted was helpful to make the motion of the arms more natural. When we first started testing the throwing motion, ATLAS tended to over-rely on the ankles joint, causing the robot to hinge and not rely heavily enough on the arms. The weighted diagonal matrix with low weights on the knees and arms relative to the ankles allowed ATLAS to make use of the joints in the arms and knees more. The formula for the damped inverse, from the Generalized Inverse handout, is listed below.

$$J_{\text{Dinv}} = (J^T J + \gamma^2 I_N)^{-1} J^T = J^T (J J^T + \gamma^2 I_M)^{-1}$$

We then calculated qc_dot by using the formula for in the Generalized Inverse handout:

$$\dot{q} = J_p^{\dagger} v_p + \left(I_N - J_p^{\dagger} J_p\right) \dot{q}_{\text{secondary}}$$

Here, since the primary velocity task is zero to keep the right foot stationary with respect to the left foot, the first term disappears. q_secondary,dot was of course calculated from the damped inverse of the jacobian for the secondary task. The weighted diagonal matrix was defined in the initialization of the code here:

```
# damping factor
        self.gamma = 0.08
        self.W = np.eye(30)
```

```
        self.w_low = 0.07
        self.w_high = 2
        arm_indices = range(15, 27)
        self.W[3, 3] = self.w_low  # knee_left
        self.W[9, 9] = self.w_low  # knee_right
        self.W[arm_indices, arm_indices] = self.w_low
        self.W[4, 4] = self.w_high  # aky_left
        self.W[10, 10] = self.w_high  # aky_right
```

And the snippet of code calculating the damped, weighted inverse and consequently the qcdot is here:

```
        W_inv = np.linalg.inv(self.W)
        J_w_inv_dls = (
             W_inv
             @ self.bigJ.T
             @ np.linalg.inv(
                self.bigJ @ W_inv @ self.bigJ.T + self.gamma**2 * np.eye(18)
             )
          )

          qcsec = (
               J_w_inv_dls
               @ np.hstack((np.zeros(12), self.right_arm.vd, self.left_arm.vd)).T
           )
          qcdot = (np.eye(30) - np.linalg.pinv(Jf) @ Jf) @ qcsec
```

One particularly difficult challenge for us was how to address the slowing down of the joints after reaching the desired position and velocity so that the motion carried through and looked natural. We accomplished this by switching the secondary task from working out the spline to finding the joint difference between the current and the start position, and using a gain to regulate how fast the convergence to the initial position would be. This had the advantage of ensuring we ended close to the start position, but unlike if we used another spline, this method does not achieve continuous velocity between switching from the spline to this task. We tried setting another gain to incorporate the joints' current velocity into the next velocity (essentially functioning as a PD controller), but this had little effect on how the joints moved, so for simplicity this was removed. A snippet of the code for this secondary task can be found below:

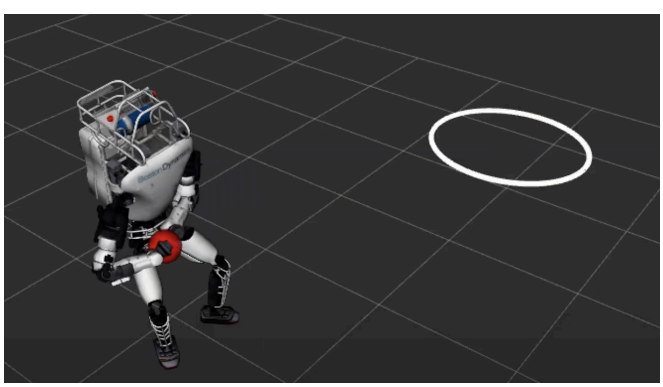
```
q_current = np.hstack(
    (
        self.left_leg.qc,
        self.right_leg.qc,
        self.left_arm.qc,
        self.right_arm.qc[3:],
        self.head.qc[3:],
    )
).T

q_rest = np.hstack(
    (
        self.left_leg.q0,
        self.right_leg.q0,
```

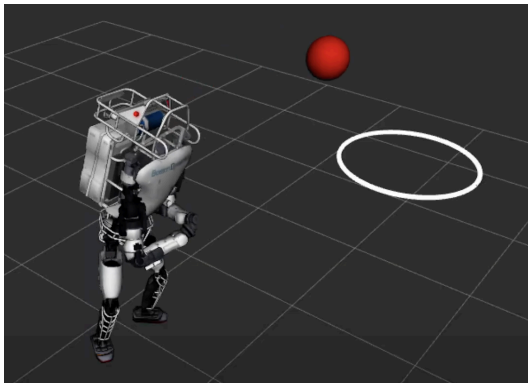
```
        self.left_arm.q0[:10],
        self.right_arm.q0[3:],
        self.head.q0[3:],
    )
).T

k_posture = 6

v_t = k_posture * (q_rest - q_current)
qcdot = (np.eye(30) - np.linalg.pinv(Jf) @ Jf) @ v_t
```

## IV. Cases and Testing

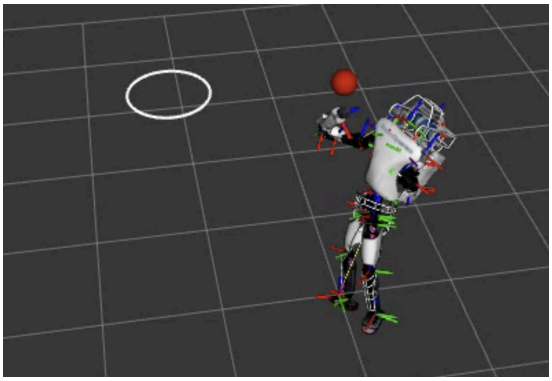
We tested four main cases of the robot's motion.

(1) *Default joint positions.* This is the simplest case.

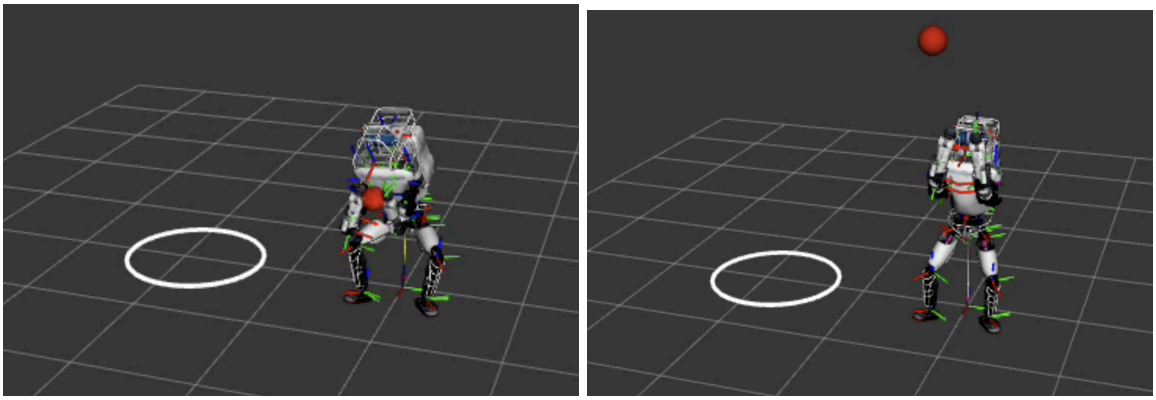




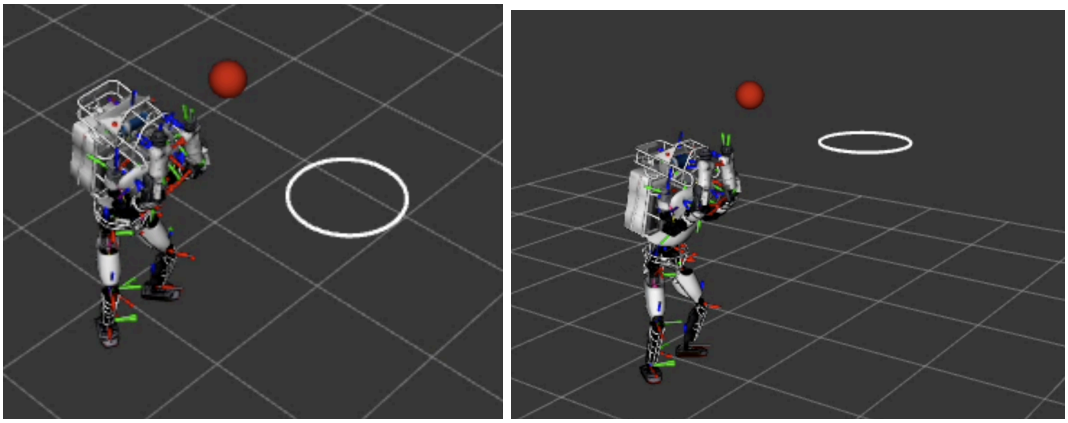
(2) *Mini-Layup.* In this case, we brought the hoop much closer and changed the initial joint angles.



(3) *3-pointer*. In this case, we placed the hoop much farther and lower than the previous two cases.



(4) *Different joint positions.* We chose different starting joint positions for the same hoop location as case 1 to test the robustness of the code.



(5) *Faulty knee.* We removed the function of the left knee and locked it to be straight.