Internship Project Report

Jyoti Paswan

(Roll no. 2022031138)

Third Year Undergraduate Student – Electrical Engineering

Madan Mohan Malaviya University of Technology, Gorakhpur

Under the guidance of

Prof. Ravi Prakash, Hi-Ro Lab, RBCCPS

Indian Institute of Science

Bengaluru, Karnataka, India

(e-mail- jyotipaswan332004@gmail.com)

Abstract

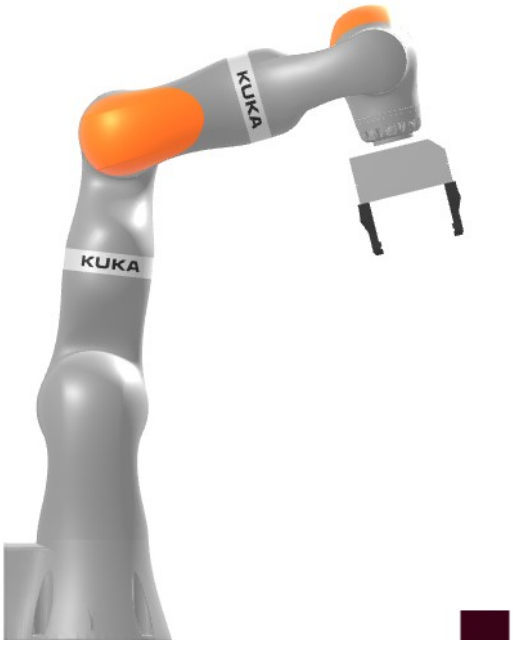
This report presents a simulation-based approach to robotic manipulation using the Drake physics engine. Two tasks were implemented: a dynamic object-throwing behavior and a pick-and-place operation. A KUKA iiwa14 robotic arm was used to perform both tasks, employing inverse kinematics to compute feasible joint configurations and time-indexed trajectories for smooth motion execution. For the throwing task, a velocity-based release strategy was implemented to simulate projectile motion toward a basketball hoop. The simulation environment was tested on both local and cloud-based platforms and visualized in Meshcat. These simulations successfully demonstrate the integration of kinematic planning, trajectory generation, and execution for both static and dynamic manipulation tasks.

1. Introduction :

1.1 What is Robotic Manipulation?

Robotic manipulation refers to the ability of a robot to interact physically with objects in its environment. This includes a wide range of tasks such as picking, placing, assembling, pushing, pulling, or even throwing. At its core, manipulation combines mechanical control, perception, motion planning, and often decision-making to allow robots to perform actions that would otherwise require human dexterity and precision.Modern manipulators, such as the KUKA iiwa arm used in this project, are equipped with multiple joints and end effectors (grippers or tools) that allow them to execute complex tasks in constrained or unstructured environments. These manipulators can operate in factories, warehouses, hospitals, or even homes, replacing or assisting humans in tasks that are dangerous, repetitive, or demand high precision.Modern robotic manipulation systems are composed of multiple subsystems working in harmony:

- **Perception systems** to detect and localize objects using sensors or cameras.
- **Planning algorithms** to compute paths or trajectories to reach desired poses.
- **Control systems** to translate planned motions into torque, position, or velocity commands.
- **Physical simulation or real-world dynamics** to predict how an object will behave after contact or release (e.g., during a throw or push).



(fig. 1 KUKA iiwa arm having 7 joints)

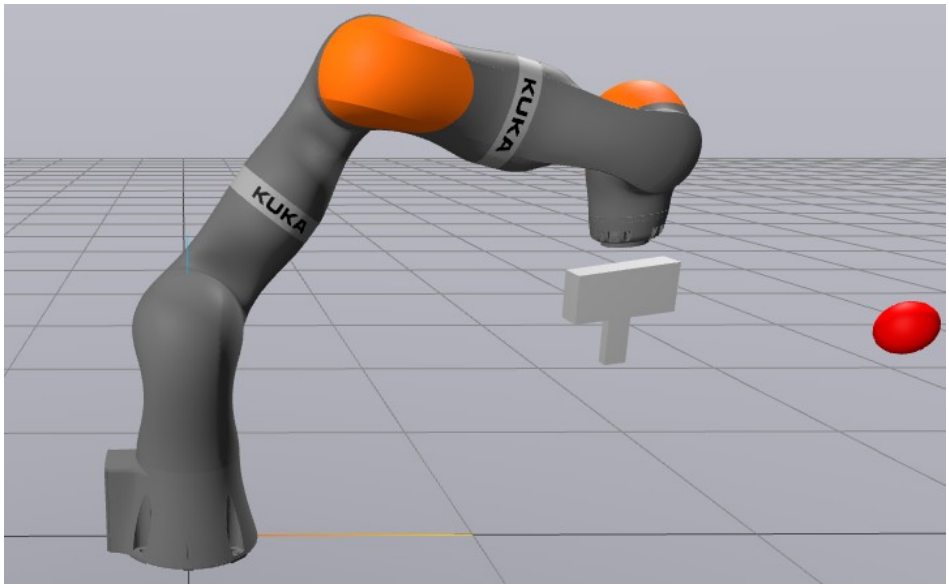
1.2 Importance of Throwing Tasks in Robotics

Throwing is an advanced form of robotic manipulation that represents a significant leap beyond traditional pick-and-place operations. In conventional manipulation, robots are limited by their reach and speed—requiring direct physical access to the start and end positions of a task. However, by learning to throw, robots can extend their operational workspace, interact with distant targets, and complete tasks in a more time-efficient and energy-efficient manner.

From a scientific perspective, robotic throwing integrates elements of motion planning, control theory, dynamics, and optimization. Unlike placing an object, throwing requires the robot to precisely control the release velocity, angle, and timing, accounting for the mass and shape of the object and the influence of gravity. The throwing motion transforms the robot's kinematic energy into the projectile motion of an object, governed by Newtonian physics. Predicting where an object will land after release requires accurate modeling of ballistic trajectories, including factors like drag, spin, and object inertia.

Furthermore, throwing introduces the concept of **underactuated manipulation**, where the robot influences an object not just during grasp but through its post-release trajectory.

In our project, throwing was implemented using the KUKA iiwa robot in simulation. The robot learned to pick up a ball and execute a dynamic motion that released the object toward a basketball hoop. This task required careful tuning of the end-effector velocity and alignment, as well as verification of trajectory prediction—all without relying on high-fidelity vision or feedback, showcasing the potential of model-based planning for dynamic manipulation.

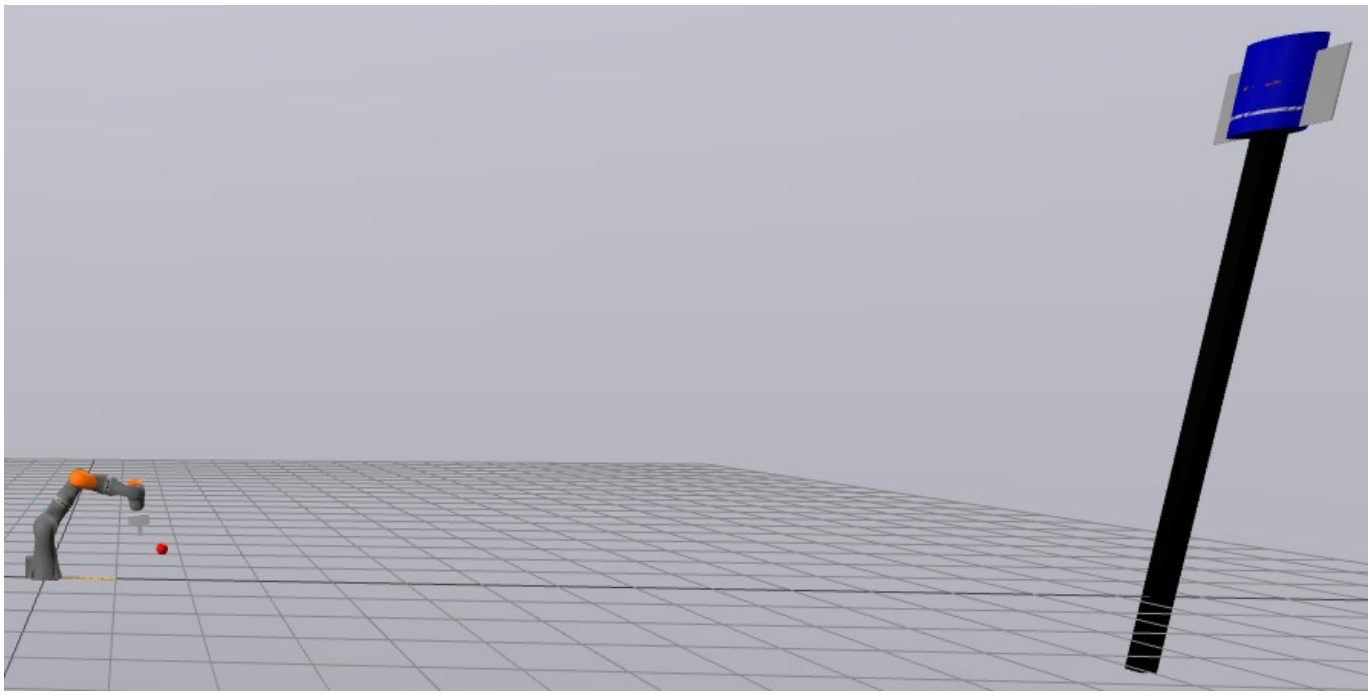


(fig. 2 Intial stage of throwing)

1.3 Introduction to Drake as a Physics-Based Simulator

Drake is an open-source robotics simulation and modeling toolbox developed by the **Toyota Research Institute** and **MIT's Robot Locomotion Group**. It is designed to simulate complex, real-world robotic systems with high physical fidelity, making it a powerful alternative to other simulators like PyBullet, Gazebo, or MuJoCo. Unlike many game-engine-based platforms, Drake prioritizes accurate physics, contact dynamics, and mathematical modeling over visual realism, making it particularly suitable for academic research, prototyping, and control system development. Drake supports the full lifecycle of robot design and experimentation—from modeling and kinematics to simulation and control—through a modular and object-oriented architecture built primarily in C++ with Python bindings for usability.

At the core of Drake is its **MultibodyPlant** framework, which enables users to construct articulated rigid body systems with constraints, actuators, and contact models. It supports collision detection, friction modeling, and dynamic simulation. For real-time visualization, Drake integrates seamlessly with **Meshcat**, a browser-based 3D viewer that allows users to observe simulations, robot poses, and trajectory playback directly in the browser. Drake also includes tools for inverse kinematics, trajectory optimization, feedback control, and sensor simulation—allowing for comprehensive experimentation within a single ecosystem. This project uses Drake extensively to model a **KUKA iiwa robotic arm**, simulate object manipulation, and test dynamic throwing behavior in a controlled virtual environment.

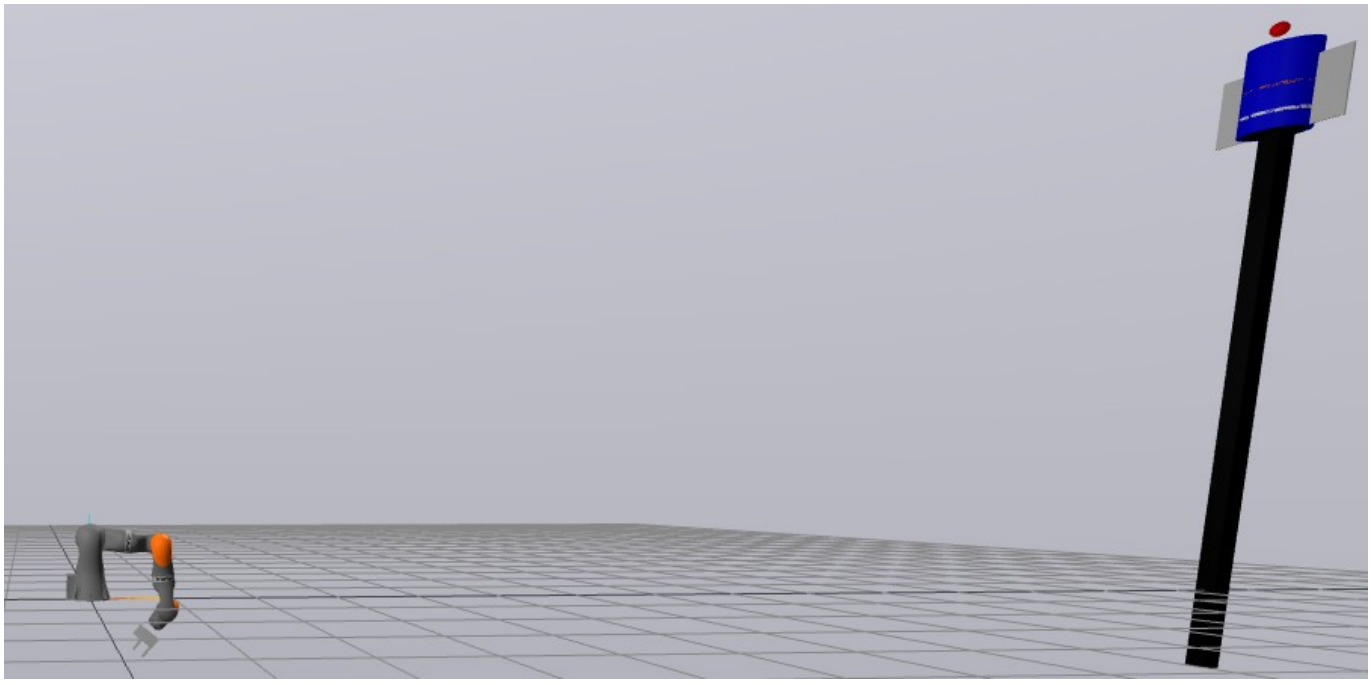


(fig. 3 Drake simulation showing KUKA iiwa, ball, hoop, and Meshcat visualization panel.)

1.4 Objective of the Project

The central objective of this project is to explore and implement robotic manipulation behaviors—most notably the simulation of a **throwing task**, followed by a **pick-and-place task**—using the **Drake robotics simulator**. By leveraging Drake's physically accurate multibody dynamics engine and inverse kinematics tools, the project aims to demonstrate that complex, dynamic behaviors such as robotic throwing can be achieved in simulation with realistic motion planning and precise control strategies. A 7-DOF (degree-of-freedom) **KUKA iiwa robotic arm** was chosen as the primary manipulator, due to its widespread use in industrial automation and academic research.

The first and more challenging goal was to simulate a **robotic throw**, where the robot picks up a spherical object (representing a ball) and throws it toward a hoop positioned at a distance, mimicking a basketball shot. This required careful design of the robot's gripper trajectory, end-effector alignment, and release velocity. Since the success of the throw depends heavily on dynamic parameters such as object mass, gravity, angular orientation, and contact timing, the project emphasized tuning these variables using Drake's trajectory tools and spatial velocity controls. This task was primarily implemented in a **local simulation environment**, where Meshcat was used to visualize the robot's arm motion and the ball's flight path. The goal was to replicate a successful throw purely through model-based reasoning—without relying on learning-based or vision-feedback controllers—thus validating the capabilities of deterministic motion planning in dynamic manipulation.

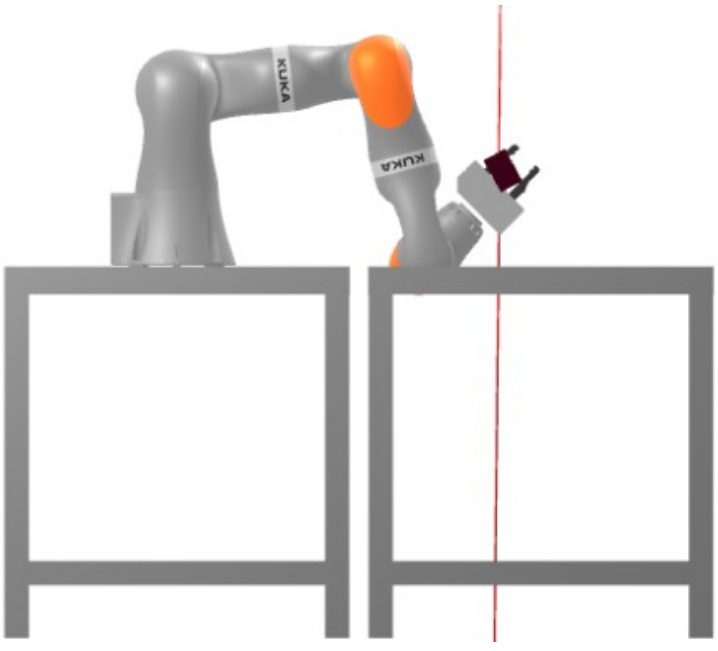


(fig.4 Meshcat visualization showing ball falling inside basketball hoop)

The second objective involved simulating a **pick-and-place task**, where the robot grasps a brick-like object from one table and transfers it to another. This was implemented using **inverse kinematics** to compute feasible joint configurations at both the pickup and placement locations. A smooth trajectory was then interpolated between these poses to ensure stable, collision-free motion. This task was designed to run on **both local and cloud-based simulation setups**, showcasing the flexibility and portability of the Drake simulation stack.

The robot grasps a brick-like object from one table and transfers it to another. This was implemented using inverse kinematics to compute feasible joint configurations at both the pickup and placement locations. A smooth trajectory was then interpolated between these poses to ensure stable, collision-free motion. This task was designed to run on both local and cloud-based simulation setups, showcasing the flexibility and portability of the Drake simulation stack. It also served as a baseline manipulation task to compare with the more dynamic throwing scenario, helping to demonstrate the continuum between quasi-static and dynamic robotic behaviors.

The simulation required precise calibration of gripper pose relative to the object's centroid to guarantee successful grasping without visual perception. Trajectories were generated using position interpolation methods and executed using `TrajectorySource` blocks in Drake's diagram builder. The task also provided an opportunity to experiment with **frame welding**, object anchoring, and gravity compensation in Drake. By validating the same pick-and-place logic across platforms, the experiment confirmed the robustness of Drake's plant-model abstraction and highlighted the practical importance of having simulation pipelines that can run identically on cloud-based and local environments.



(fig. 5 KUKA iiwa robotic arm grasping the brick)

2. Tools & Technologies Used

2.1 Drake: Features, Installation, and Capabilities

Drake is the central simulation framework used in this project. Developed by the Toyota Research Institute and MIT, it provides a robust platform for modeling and controlling complex robotic systems. Its features include:

- Multibody dynamics modeling using `MultibodyPlant`
- Support for **inverse kinematics (IK)**, **trajectory planning**, and feedback control
- Accurate contact modeling with Coulomb friction and restitution
- Integrated tools for geometry visualization, collision checking, and physics-based simulation
- Support for both **symbolic** and **numerical** computations

Drake is installed using the official `pip` distribution, which ensures compatibility with Python-based scientific computing environments. The installation in this project used:

```bash
pip install drake
```

Drake's modular architecture allows users to build complex systems as **composable diagrams**, where each subsystem (e.g., controller, visualizer, planner) is a block connected via ports. This enables clean separation of concerns and debugging of each component independently.

(Code snippet)

```python
builder = DiagramBuilder()
plant, scene_graph = AddMultibodyPlantSceneGraph(builder, time_step=0.001)
parser = Parser(plant)
parser.AddModelFromFile("iiwa14.urdf")
plant.Finalize()
```

2.2 Meshcat: Real-Time Web Visualization

Meshcat is a lightweight, browser-based 3D visualization tool that is tightly integrated into the Drake ecosystem. It provides an intuitive and interactive interface for observing the spatial behavior of robotic systems in real time. Meshcat offers fast performance, minimal setup overhead, and live control from Python code, making it especially suitable for rapid development and debugging.

Meshcat operates by exposing a local or remote web server that hosts a 3D scene rendered using WebGL. Within this environment, users can view robot poses, animation playback, object transformations, joint angles, and even real-time collision geometries. The camera controls are fully interactive, allowing users to zoom, pan, and rotate around the scene with ease. This real-time interactivity is particularly valuable when designing and validating motion plans—such as watching a robotic arm release a ball mid-throw or verifying that a gripper reaches the correct target frame during a pick operation.

In this project, Meshcat was used extensively to debug robot kinematics, validate inverse kinematics solutions, and visually confirm throwing trajectories. For instance, it allowed the user to observe how variations in end-effector velocity changed the parabolic arc of the thrown object. It was also helpful in fine-tuning the relative positions of the robot, the tables, the ball, and the basketball hoop by providing immediate visual feedback. Because Meshcat works directly with Drake's `geometry::SceneGraph`, it automatically reflects all changes in robot configuration, including pose updates and object spawning.

(Code Snippet)

```python
from pydrake.visualization import StartMeshcat
meshcat = StartMeshcat()
meshcat.SetObject("world/target", Sphere(0.05), Rgba(1, 0, 0, 1))
```

2.3 Python Libraries and Dependencies

In addition to Drake's native Python bindings, this project made extensive use of several standard scientific computing libraries from the Python ecosystem. These libraries were critical for numerical operations, trajectory analysis, vector transformations, and visualization.

- **NumPy** (`numpy`): Used for matrix and vector operations throughout the project. NumPy arrays were essential for computing spatial transforms, joint angles, trajectory interpolation, and object positions. Since Drake functions often return or expect NumPy-compatible data structures, the use of NumPy was both convenient and necessary for high-performance numeric computation.

  Ex : Creating a position vector

  ```python
  position = np.array([x, y, z])
  ```

- **SciPy** (`scipy.spatial.transform`): Specifically, the `Rotation` class was used to convert between rotation representations—such as Euler angles, rotation matrices, and quaternions. This was especially important for configuring the gripper orientation during the throwing task, where precise alignment of the end-effector was critical to ensure proper trajectory.

  Ex : Creating a rotation matrix from Euler angles.

  ```
  R = Rotation.from_euler("xyz", [roll, pitch, yaw]).as_matrix()
  ```

- **matplotlib**: Employed to visualize numerical data such as joint trajectories, end-effector velocities, and object positions over time. While Meshcat provides 3D visualization of the scene, `matplotlib` was useful for creating analytical plots for debugging and report generation.

  Ex : Plotting ball velocity over time.

```
plt.plot(time_array, velocity_array)
plt.xlabel("Time (s)")
plt.ylabel("Velocity (m/s)")
plt.title("Throwing Velocity Profile")
```

IPython.display: Used within Jupyter notebooks (especially in the cloud setup) to embed HTML and Meshcat viewers inline. This helped create a compact, interactive simulation environment during cloud-based experimentation.

Drake's native modules: These include:

- `pydrake.all` for general access to Drake's modeling and visualization tools
- `pydrake.trajectories` for building time-indexed trajectories
- `pydrake.multibody.plant` for robot modeling and contact simulation
- `pydrake.geometry` for shape creation and object registration.

All project dependencies were managed and installed using Python's standard package manager, pip, ensuring compatibility with virtual environments and platform-agnostic execution.

The Python libraries used are fully compatible with CPython 3.8, the version required by the pip-distributed Drake 1.26.0 build. Dependency management was consistent across both local (Ubuntu-based) and remote (Jupyter/Colab) environments, allowing for seamless portability of code and reproducibility of simulation results.

The integration of Drake's Python API with NumPy, SciPy, and `matplotlib` enabled a highly modular development process. Vectorized operations, matrix transformations, and trajectory generation could be tightly coupled with Drake's simulation loop, allowing for deterministic control over multibody plant states, pose estimation, and time-indexed inputs. This alignment between numerical computation libraries and Drake's symbolic-numeric hybrid framework accelerated prototyping, facilitated precise debugging, and ensured that control algorithms could be validated efficiently within the simulation architecture.

3. Simulation Environment

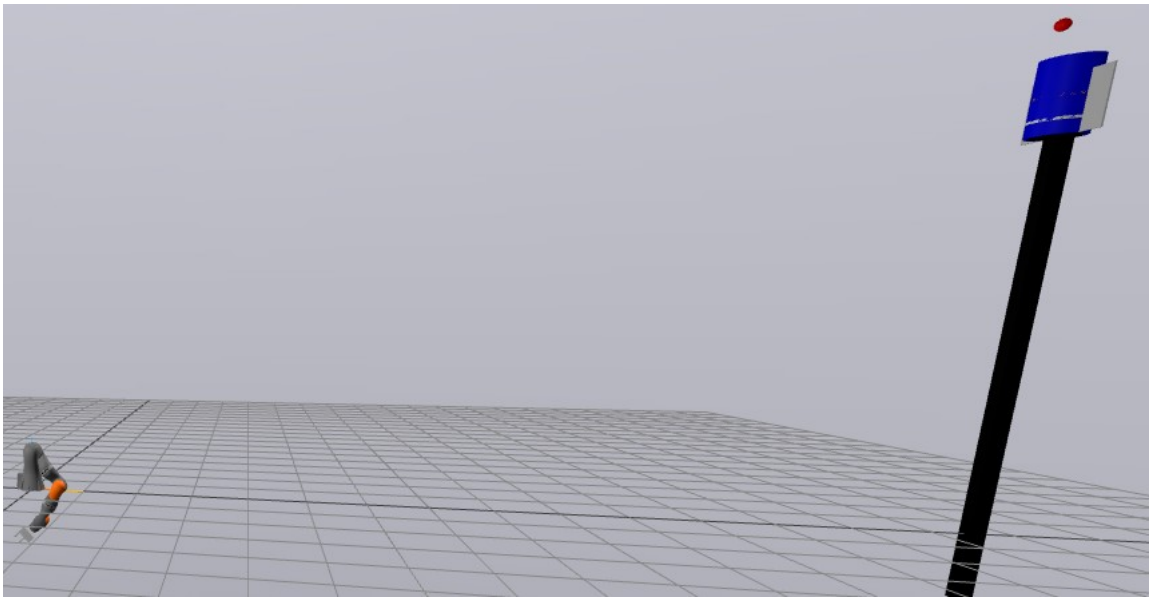
3.1 Robot and Scene Setup

The primary manipulator used in this project is the **KUKA iiwa14**, a 7 joints industrial robotic arm widely used for precision manipulation in research and automation. The arm provides a redundant configuration, allowing for multiple feasible joint poses to reach a single end-effector target. In this simulation, the KUKA iiwa14 model was loaded using Drake's URDF parser and integrated into the `MultibodyPlant` system.

The robot was equipped with a simplified **parallel jaw gripper**, modeled as a rigid extension at the end-effector link. The gripper was either welded directly to the end-effector frame or controlled via additional joints (depending on the variant), but in this project, the gripper was treated as a rigid, passive body to reduce simulation complexity. This configuration was sufficient for both the pick-and-place and throwing tasks.

The **basketball hoop** was placed on the target table, modeled using cylindrical primitives to create the hoop's ring and supporting pole. Proper alignment between the hoop and the throwing arc required accurate positioning in world space, often involving the use of midpoints, trigonometric calculations, and vertical offsets to place the ring at a realistic height.

To handle frame transformations, all major objects—robot, ball, hoop, and tables—were assigned explicit RigidTransforms between their local frames and the world frame. These transforms defined both translation vectors and rotation matrices, allowing precise control over object placement and relative orientations. In the throwing task, the gripper's final frame before release



(fig. 6 Meshcat showing the full setup: robotic arm,tennis ball and basketball hoop)

3.2 Object Properties

Ball (Throwing Task)

The object used in the throwing task was modeled as a **solid sphere**, created using Drake's `Sphere` geometry. The radius was chosen to approximate a small basketball (e.g., 0.06 m), and its mass was set to match realistic lightweight values (e.g., 0.1–0.2 kg). The ball was added to the simulation as a **free-floating rigid body**, allowing it to be manipulated dynamically via velocity commands and released by the gripper mid-motion.

To simulate a throw, the robot was programmed to assign a **spatial velocity** to the ball upon release. This was achieved by detaching the ball from the gripper and assigning it a velocity in world coordinates using:

(Code Snippet)

```python
plant.SetFreeBodySpatialVelocity(ball_body, SpatialVelocity(...))
```

Hoop (Target)

The hoop was constructed using Drake's `Cylinder` geometry to approximate a standard basketball ring. A horizontal cylinder represented the hoop itself (with a radius ~0.15 m), and a vertical cylinder served as the support pole. Both were welded to a base box (or table) to ensure they remained static during simulation.

The target pose of the hoop was carefully chosen to match the expected parabolic arc of the ball based on velocity and release height. This required iterative tuning of both object positions and initial velocities during testing. The hoop's open side was oriented to face the robot's gripper to maximize throw success.

(Code Snippet)

```
hoop_cylinder = Cylinder(radius=0.15, length=0.02)
plant.RegisterCollisionGeometry(..., hoop_cylinder, CoulombFriction(0.9, 0.8))
```

Gripper and Constraints

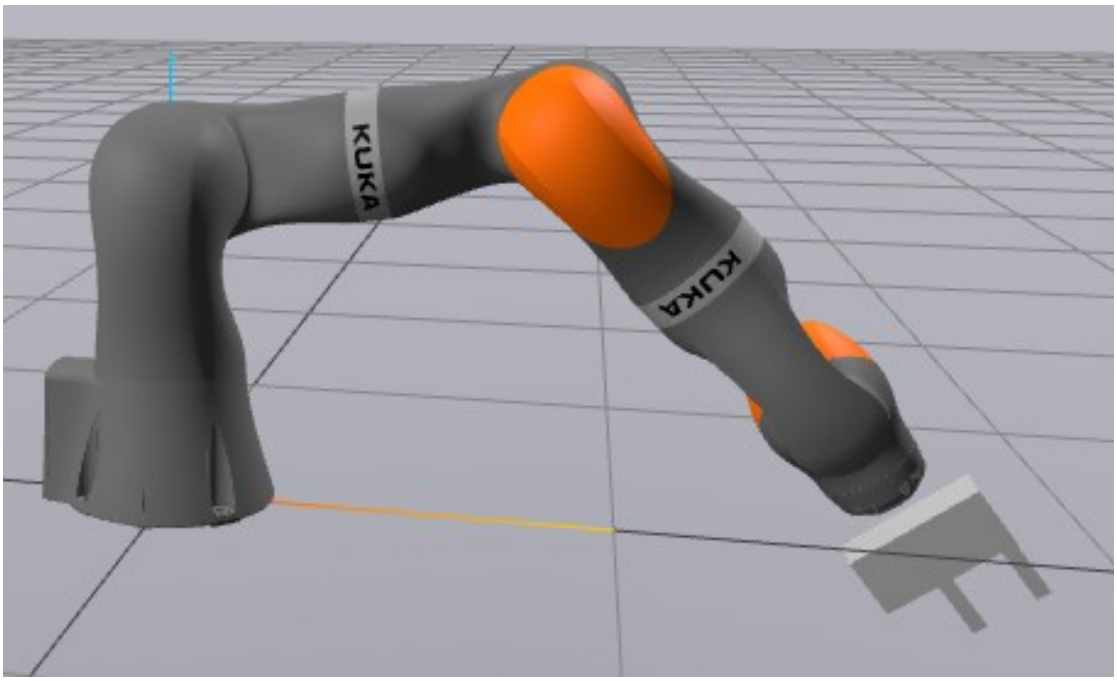
The gripper was modeled either as part of the robot or as an attached rigid body. In the pick-and-place task, the object was "grasped" by **welding** the object frame to the gripper frame using:

(Code Snippet)

```
plant.WeldFrames(gripper_frame, object_frame, RigidTransform.Identity())
```

For throwing, the object was initially welded or placed into position, and then the weld was removed programmatically using `plant.ClearWeld()` before assigning a velocity. This allowed the ball to follow a free-fall trajectory under gravity once released from the robot.

The simulation assumed rigid grasping without slip or finger contact modeling. This simplification was valid for the objectives of the project, which focused more on pose and velocity planning than on contact dynamics.



(fig. 8 welded gripper in the KUKA robotic arm)

4. Inverse Kinematics & Trajectory Planning

In this project, the robot needs to **move its arm in a smart way** to throw a ball into a hoop. To do this, we used two key techniques:

1. **Inverse Kinematics (IK)** — to figure out how the robot should aim and where its joints should be.
2. **Trajectory Planning** — to make the robot arm move smoothly and correctly through those poses.

4.1 Inverse Kinematics (IK)

Inverse Kinematics is the process of finding joint angles for a desired position and orientation of the robot's end-effector (in our case, the gripper).

We want the gripper to:

- Reach a certain position in 3D space (close to the ball or aimed toward the hoop),
- Point in the correct direction (usually toward the hoop when throwing).

We define this desired pose using a **RigidTransform** in Drake, which includes both position and orientation.

(Code Snippet)

```python
# Define desired direction to aim at the hoop
direction = hoop_position - gripper_position
unit_direction = direction / np.linalg.norm(direction)

# Create a rotation matrix using that direction
rot = RotationMatrix.MakeFromOneVector(unit_direction, axis='z')
pose = RigidTransform(rot, gripper_position)
```

We then use **Drake's InverseKinematics solver** to compute the joint angles needed for this pose.

```
ik = InverseKinematics(plant, context)
ik.AddPositionConstraint(...)        # Keep position close to desired
ik.AddOrientationConstraint(...)     # Align gripper with hoop
result = Solve(ik.prog())
q_throw = result.GetSolution()
```

4.2 Trajectory Planning

Once we have the joint angles for the throw pose, we need to **move the robot from its current pose to this target pose smoothly**.

This is done using a **cubic trajectory**, which guarantees smooth position and zero velocity at the start and end (no jerky motion).

Let:

- q0: starting joint angles,
- q1: target throw joint angles,
- T: duration (e.g., 1.3 seconds).

We build a trajectory like this:

```
from pydrake.trajectories import PiecewisePolynomial

q_traj = PiecewisePolynomial.CubicWithEndVelocity(
    [0.0, T],              # time range
    [q0, q1],              # joint angles
    [np.zeros(7), np.zeros(7)]  # zero velocities
)
```

4.3 Tuning

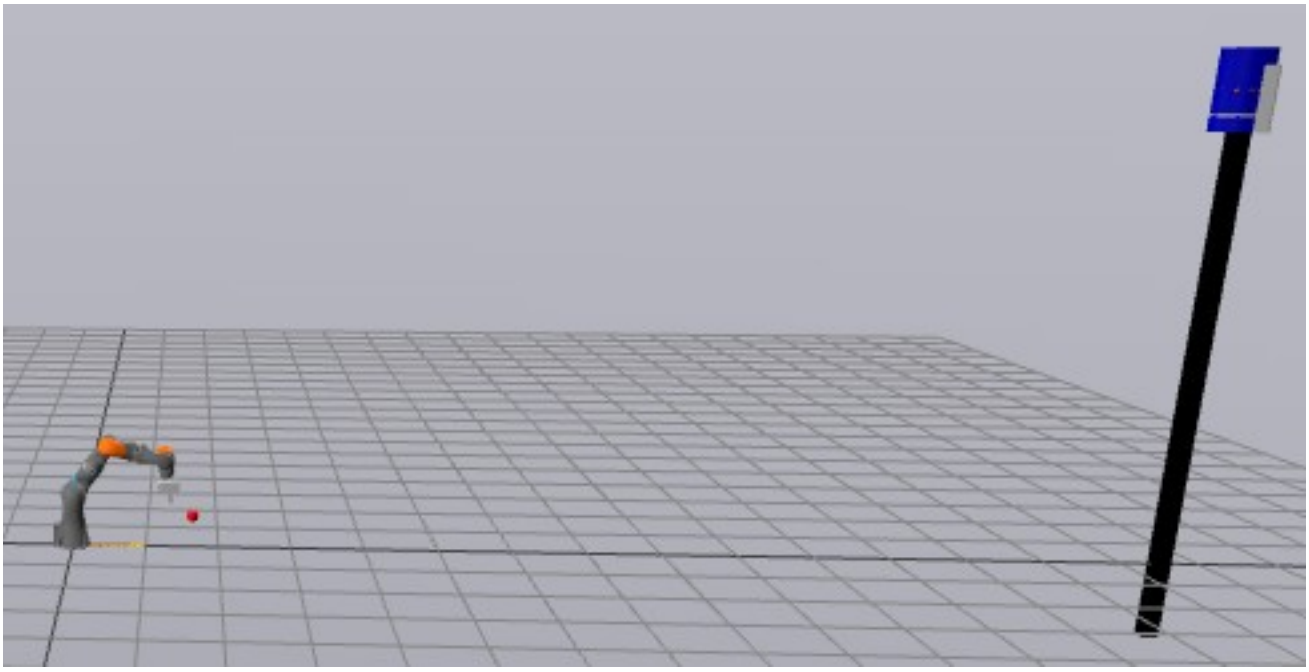
To make consistent throws, we iteratively experiment with:

- **Orientation tweaks**: small joint angle offsets (±5°),
- **Timing adjustments** T,
- **Velocity adjustments** v.

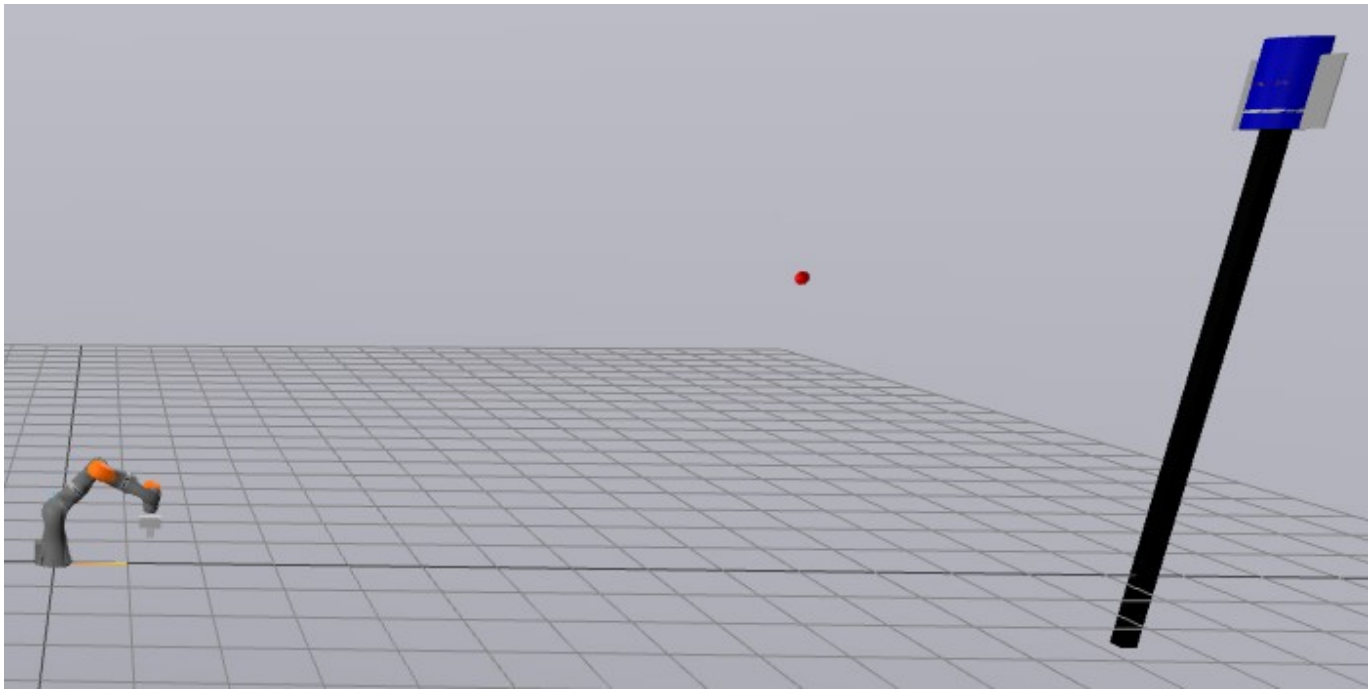
Example tuning data:

Trial	Pitch Offset	T (s)	v (m/s)	Result
1	0°	1.2	3.0	Short
2	+5°	1.2	3.2	Hook shot
3	+5°	1.3	3.4	Success

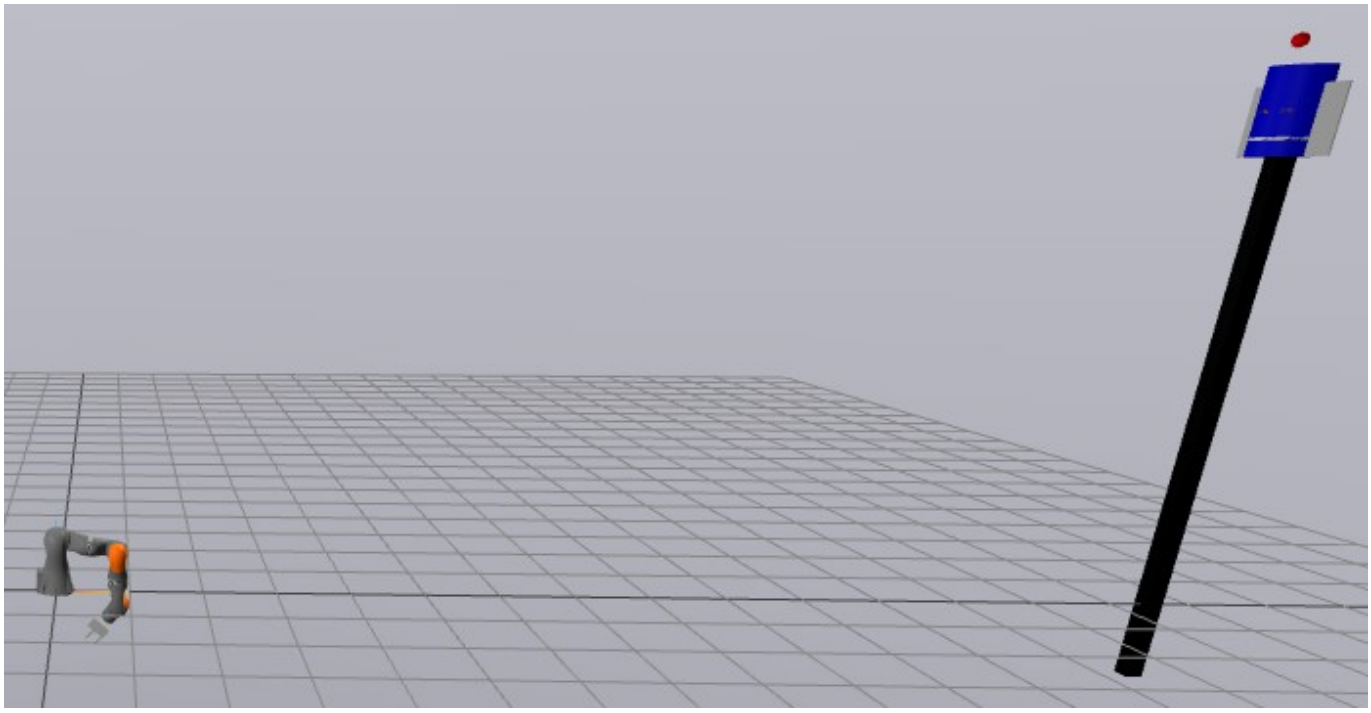
5. Results :



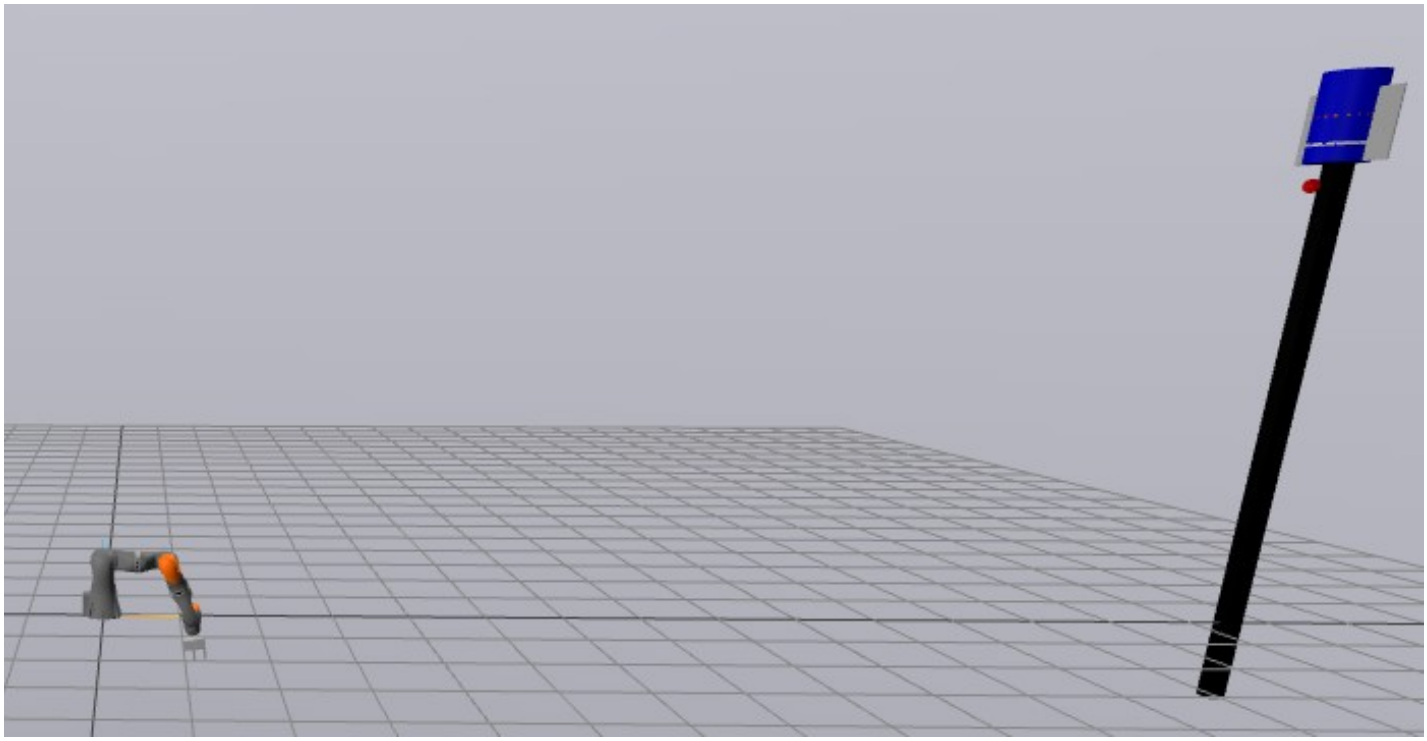
(fig.9 Initial condition when ball is on the ground)



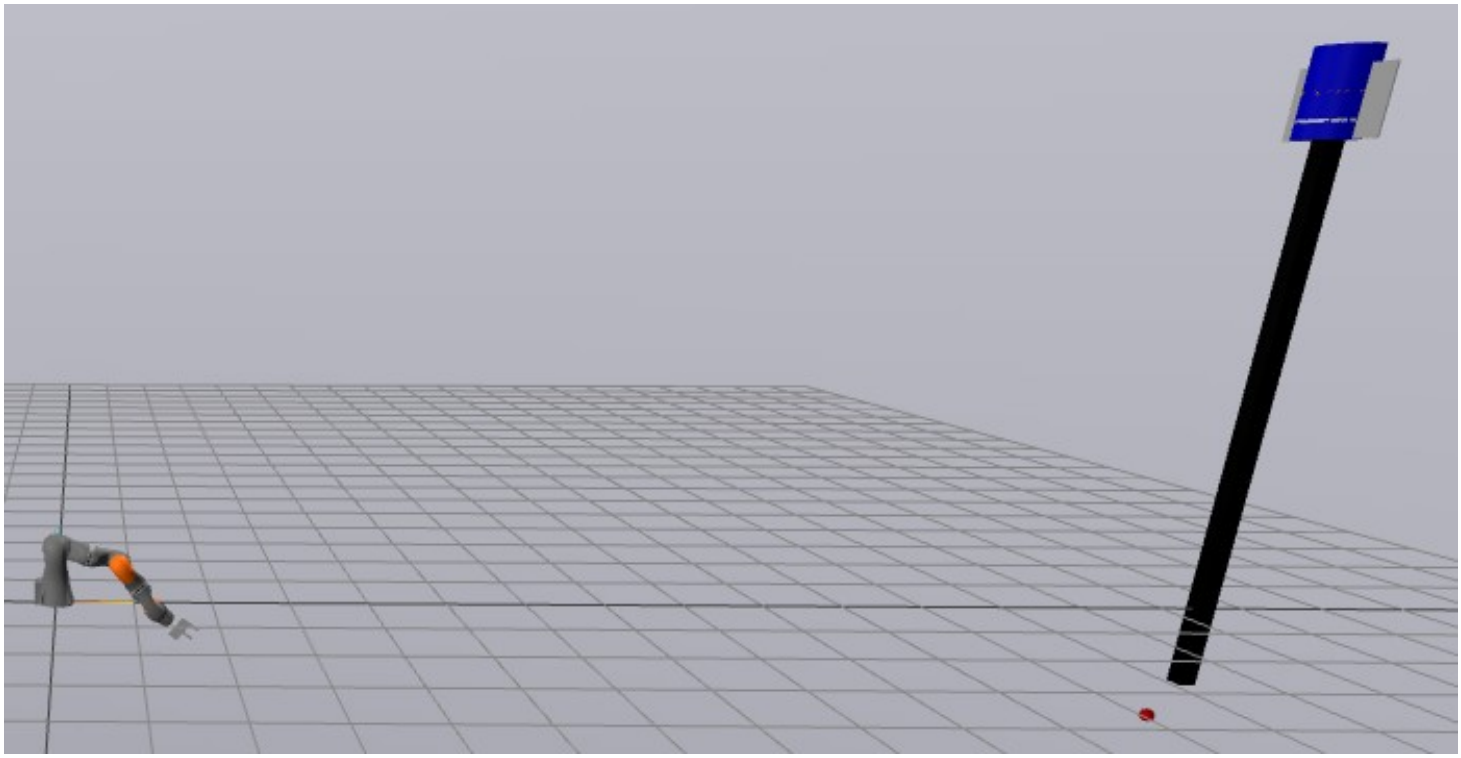
(fig.10 Ball mid-air after release with velocity arrow)



(fig. 11 Ball falling into the hoop)



(fig. 12 Ball falling out from the hoop)



(fig. 13: Final stage where the ball falls to the ground after hitting the hoop)