EYE-IN-HAND IBVS SIMULATION USING ARUCO TAGS

Qian Zilling, Atanda Abdullahi Adewale, Nauman Shafique Hashmi

Objective:

The aim of this VIBOT MSFT course project is to implement or simulate an eye-in-hand image-based visual servo control in Gazebo using a Doosan M0609 robotic arm manipulator with a camera mounted on its end effector. The visual features will be provided from 4-points in an Aruco tag.

Dependencies:

This projects runs in ROS noetic, Opencv-python, Move-it! and Gazebo.

PART 1

Robot Structure Definition, Add components and Gazebo Scene Setup:

main.xacro

- Build and configure the robotic structure for the Doosan M0609 robotic arm using URDF and Xacro.
- Add a onrobot_rg2 gripper to the end effector.
- Add the Camera

```
< robot >

< !-- Define customizable properties -->
...
< !-- Assign property values for clarity -->
...
< !-- Include relevant Xacro files for modularity -->
```

```
< !-- Include the onrobot_rg2 macro for gripper components -->
< xacro:onrobot_rg2 prefix="my_robot_rg2"/>

< !-- Define the world link -->
< link name="world" />
< !-- Create a fixed joint between the world and the robot's base -->

< !-- Create a fixed joint representing the robot's wrist (effector) -->
< joint name="wrist" type="fixed">
  < origin xyz="0 0 0" rpy="0 0 0"/>
  < parent link="link6"/>
  < child link="my_robot_rg2onrobot_rg2_base_link"/>
< /joint>

< /robot>
```

camera.xacro

- Defines a camera link (camera_link) as a visual box with a specified size and a red color for Gazebo simulation.
- Integrate a camera at the end effector joint. Creates a fixed joint (camera_joint) to rigidly attach the camera link to its parent link ("link6")
- Configures the camera to simulate a camera sensor, then define a ROS Gazebo plugin interfaces for the simulated camera for information about camera topics and distortion parameters.

```
< robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="kinova">

< !-- Define customizable properties -->
< link name="camera_link"> ...
```

< joint name="camera_joint" type="fixed"> < parent link="link6"/> < child link="camera_link"/> < origin rpy="$M_PI/2${-M_PI/2} 0" xyz="$offset_from_link_x}$ {offset_from_link_y} ${offset_from_link_z}"/> < axis xyz="1 0 0" /> < /joint> < gazebo reference="camera_link"> < sensor type="camera" name="camera_camera_sensor"> ... < plugin name="camera_camera_controller" filename="libgazebo_ros_camera.so"> ... < /plugin> < /sensor> < /gazebo> < /robot>

generatear.py

arucoworld.world

- This file generates differents 6x6 aruco tags [0-29] then plces them in the .gazebo/models folder
- This will be accesible in gazebo with a drag and drop operation
- We save the gazebo scene containing Aruco to visual_servoing/world/arucoworld.world file for easy reuse

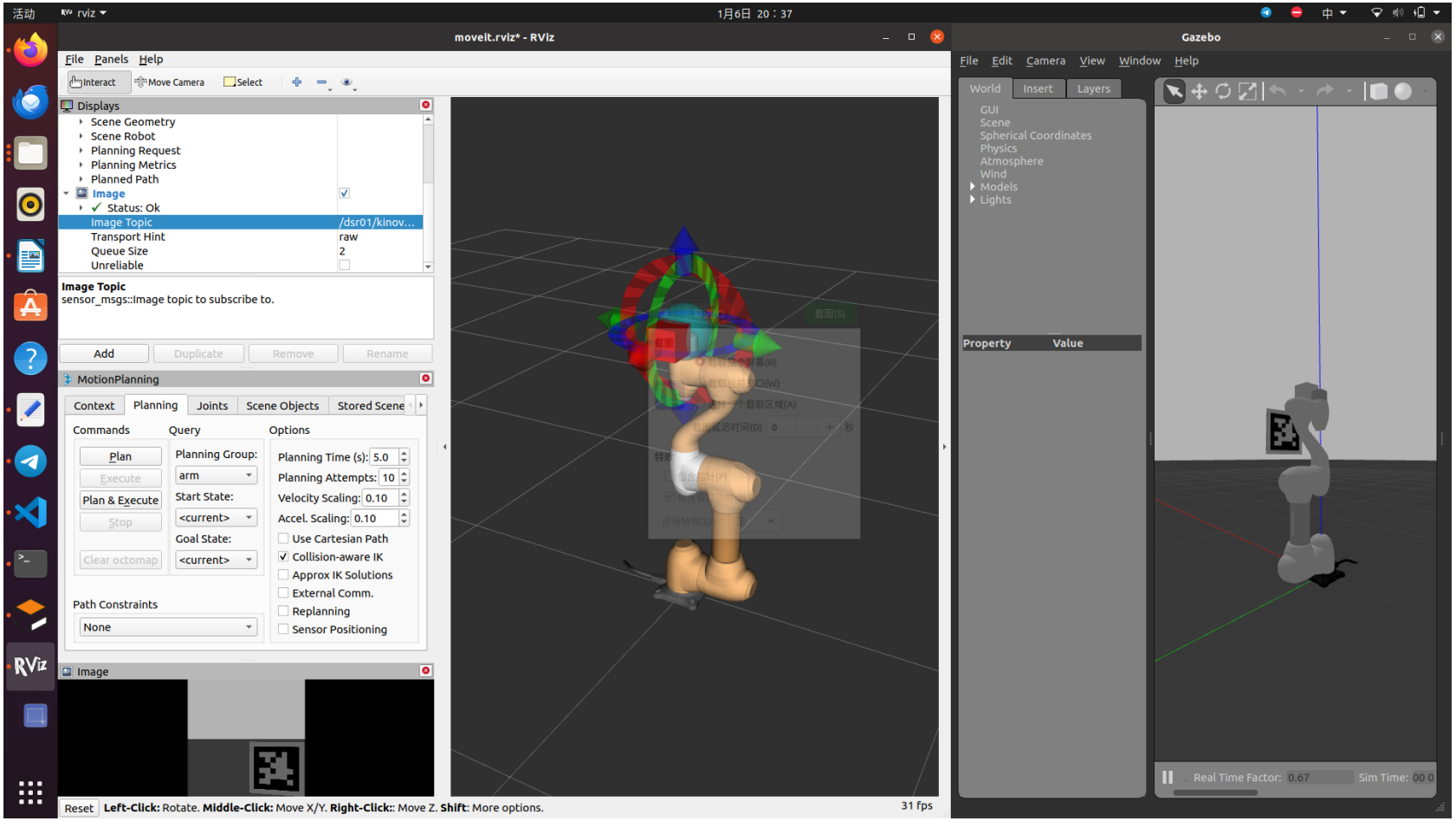
<img src="images/ar.jpg" width="900"

PART 2

Move-it and Motion Planning

dsr_moveit_gazebo.launch

- This is a modified Move-it main launch file
- It calls the default doosan move-it packages and config files to manage the movement of the m0609 robot arm.



PART 3

Camera Image processing, Aruco Tag Detection, 4 corner points Extraction, Coordinate Transformation, Moveit Integration and Visual servoing, manipulate the robot in Cartesian space:

version 1 --> viso.py

This realized the DLT method to localize the robot. It used solve PnP to get the aruco position, then we use moveit to move the end-effector link of our robot to the desired position. It’s like parking a car in the correct position.

We have 2 callback functions:

image_callback

- Subcribe to gazebo_camera Topic at **'/dsr01/kinova/camera/image_raw/compressed'** to get raw image
- Convert Image Format to OpencV format and grayscale
- ArUco Detection **cv2.aruco.Detector** using camera and detector parameters (intrinsic matrix, distortion coefficients, marker_length etc)
- Extract the 4 corner points of the detected Aruco tag with **detector.detectMarkers(gray)**

```
self.corners_list, self.ids, self.rejectedImgPoints  = detector.detectMarkers(gray)
```

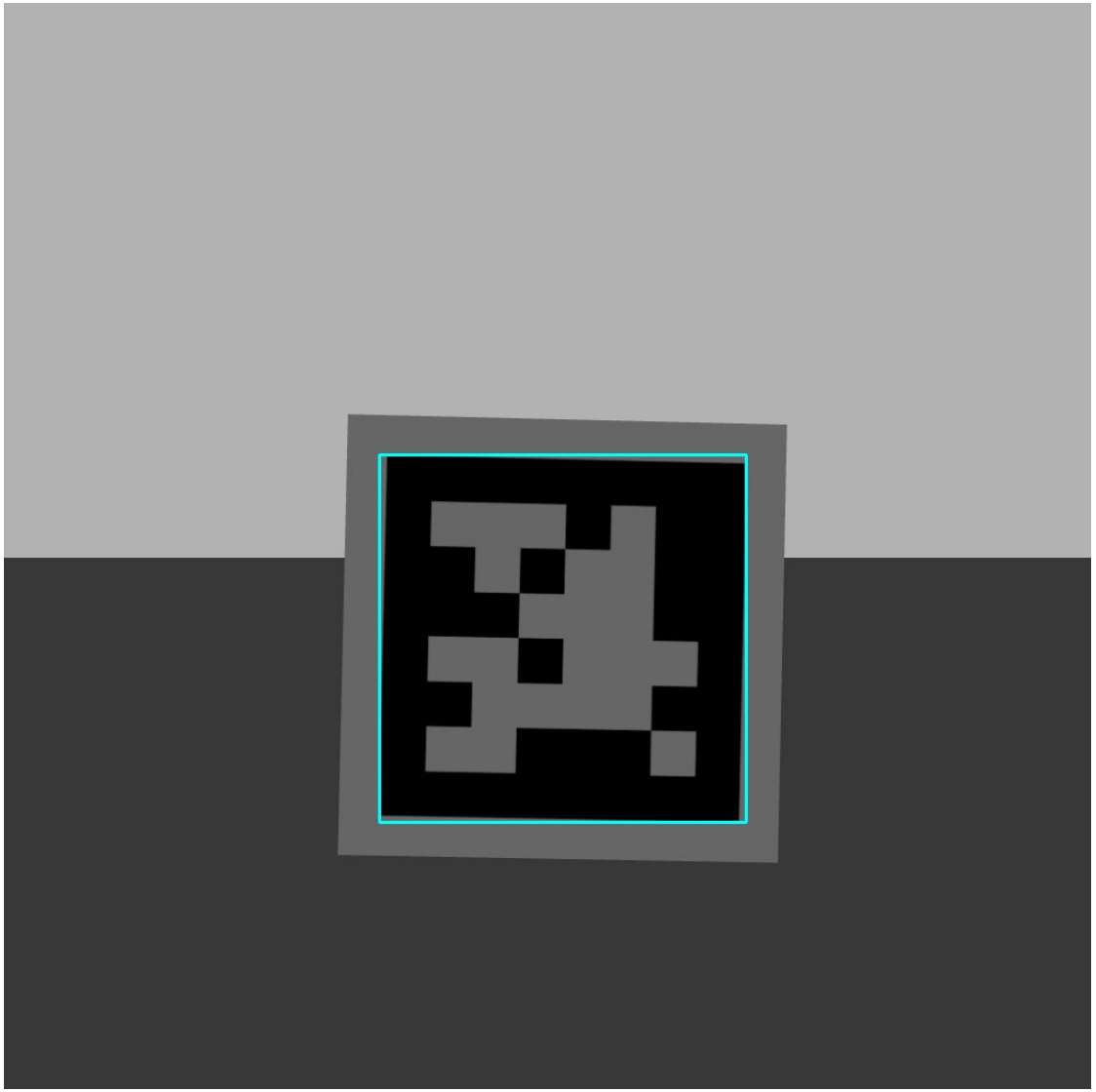
- Estimates the relative position and pose (rotation and translation) of each detected marker.
- Processes the marker information, Calculates the average movement and Publish the relative position information to topic **'/movement'** Trigger the second callback function

my_estimatePoseSingleMarkers performs Coordinate Transformation i.e transform the ArUco tag's corner points to align them with the end effector's coordinate system.

- Defines the 3D coordinates of marker corners in a known marker coordinate system.
- For each detected marker, uses solvePnP to estimate its rotation and translation vectors, appends and returns the calculated vectors.

process_ar(self, corners_list, frame) function visualizes the detected ArUco markers on the input image frame.

- Draws rectangles around the detected markers on the input image frame.



target_callback

- Sets the target of end_effector_link and plan the movement of arm,and move the arm usig moveit!.
- **moveit_commander** to integrate MoveIt! i.e to initialize and interact with MoveIt components for motion planning and control moveit_commander is a Python API provided by the MoveIt library to interact with MoveIt using Python scripts. It acts as a convenient wrapper around the MoveIt functionality, allowing users to control and plan motions for robotic arms.

```
moveit_commander.roscpp_initialize(sys.argv) < !-- Initializes the ROS C++ API.-->
< !-- moveit_commander recieves robots parameter in ROS parameter server as self.robot,  -->
self.robot = moveit_commander.RobotCommander(robot_description="/dsr01m0609/robot_description", ns = '/dsr01m0609')

< !--read the control group in the ROS param server as self.arm -->
self.arm = moveit_commander.MoveGroupCommander(name="arm",robot_description="/dsr01m0609/robot_description", ns = '/dsr01m0609')

< !-- Retrieves the end effector link of the robot arm. -->
self.end_effector_link = self.arm.get_end_effector_link()

< !-- Sets the reference frame for motion planning to "base_0." -->
self.reference_frame = 'base_0'
...
...
...
```

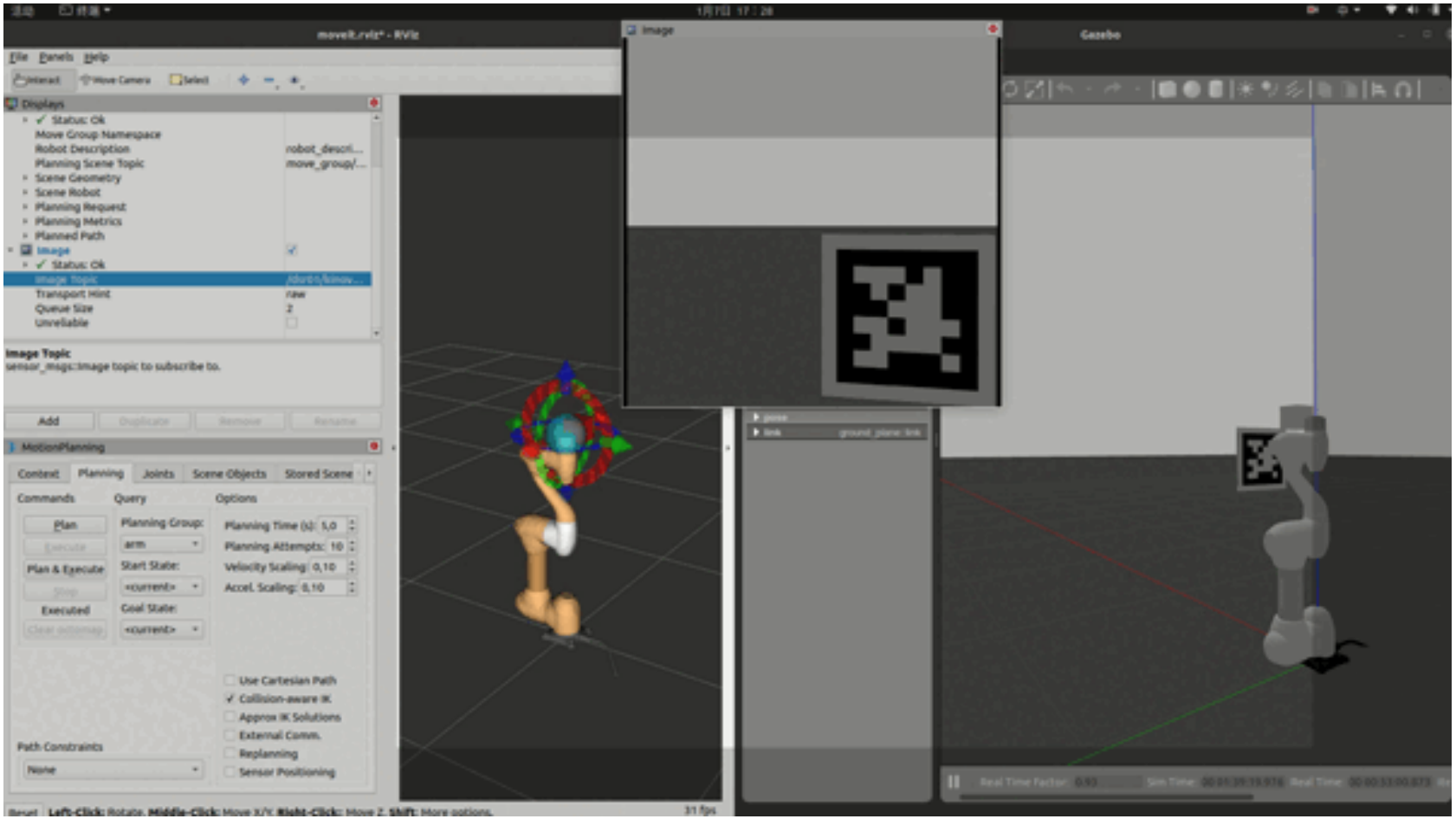
- Use the computed transformations to generate a trajectory for the robot's end effector.

```
def target_callback(self, msg):
  self.detector_enabled = False
  movements = msg.position
  current_position = self.arm.get_current_pose().pose  # Get current position

  # Adjust the target position relative to the current robot arm position
  self.target.position.x = current_position.position.x + movements.x
  self.target.position.y = current_position.position.y + movements.y
  self.target.position.z = current_position.position.z + movements.z

  # ... (other code)

  if target_x != current_x or target_y != current_y or target_z != current_z:
      # Set the target pose for the robot arm
      self.arm.set_pose_target([target_x, target_y, target_z, 0.98, 0, 0, 0])

      # Plan and execute the trajectory to reach the target pose
      self.arm.go()

      print('Target position reached!')
  else:
      print("Current state is equal to the goal state")
```

In summary, the **self.arm.set_pose_target** method sets the target pose, and the **self.arm.go()** method is used to generate a trajectory and move the robot's end effector along that trajectory to reach the target pose. MoveIt takes care of the details related to motion planning and execution.

Launch the Gazebo ans Run the ROS nodes and observe the robot's movement in response to the Aruco tag.

```
roslaunch m0609_moveit_config dsr_moveit_gazebo.launch
roslaunch visual_servoing vsm.launch
```



```
Please remove the parameter /ur01m0009/robot_description_kinematics/arm/kinematics_solver_atte
[INFO] [1704644922.145069, 5957.599000]: Seen an image
[ INFO] [1704644923.231288056, 5958.631000000]: Ready to take commands for planning group arm.
[INFO] [1704644923.235655, 5958.617000]: succeed to load robot's parameters
[INFO] [1704644923.288796, 5958.679000]: (1020, 1020, 3)
hahaha (1, 4, 2)
[INFO] [1704644923.292230, 5958.679000]: Seen aruco!
[INFO] [1704644923.323627, 5958.706000]: Seen an image
[INFO] [1704644923.358062, 5958.739000]: (1020, 1020, 3)
hahaha (1, 4, 2)
[INFO] [1704644923.362262, 5958.746000]: Seen aruco!
[INFO] [1704644923.561531, 5958.938000]: Seen an image
[INFO] [1704644923.594861, 5958.967000]: (1020, 1020, 3)
hahaha (1, 4, 2)
[INFO] [1704644923.601728, 5958.973000]: Seen aruco!
[INFO] [1704644924.612000, 5959.944000]: Seen an image
[INFO] [1704644924.653261, 5959.979000]: (1020, 1020, 3)
hahaha (1, 4, 2)
[INFO] [1704644924.658501, 5959.983000]: Seen aruco!
[INFO] [1704644925.647601, 5960.933000]: Seen an image
[INFO] [1704644925.697716, 5960.962000]: (1020, 1020, 3)
hahaha (1, 4, 2)
[INFO] [1704644925.701371, 5960.967000]: Seen aruco!
Average Movement:
 x: 0.14140003530034914
y: -0.1784388510873498
z: -0.1814360060319245
[INFO] [1704644925.729702, 5960.993000]: vision task finished!
Current position:  x: 0.0001411023641279004
y: 0.006383264930282239
z: 0.9999999679665117
START POSITION position:
  x: 0.0001411023641279004
  y: 0.006383264930282239
  z: 0.9999999679665117
orientation:
  x: 5.511259000140754e-09
  y: 8.587125946808263e-05
  z: -0.00020990302466427057
  w: 0.9999999742834232
TARGET POSITION position:
  x: 0.14154113766447704
  y: -0.17205558615706756
  z: 0.8185639619345872
orientation:
  x: 0.0
```

version 2 --> viso_follow.py

This uses the Jacobian matrix to calculate velocity from the error vector: [u-u_star,v-v_star]. We didn't find how to set the velocity, so we simulate the process still using target position control method. We assume each iter means one second in real world.

We have 2 call_back functions,but the difference here is in the first;

image_callback

Extract depth information from the ArUco marker's translation vector (tvec) obtained through the pose estimation, Z is then used in the Jacobian matrix calculation then obtain jacobian matrix and the 2d image error to calculate the relative position and rotation of the aruco tag in each iteration(i.e each second) and publish to the **'/movement'** Ros service. Since Ros services are Synchronous.

Depth Z, Calculation of Jacobian, Computation of error vector (self.e)

```
self.Z = tvec[0][-1].item()
self.Jacobi = np.linalg.pinv(self.Jacobi) (line 119)
self.e =np.array([[self.u_star-self.u],[self.v_star-self.v]]) (line 121)
```

Velocity computation and Iterative loop:

```
self.movement_temp = self.lambd * self.Jacobi @ self.e (line 123)
while np.linalg.norm(self.e) >= self.threshold: (line 132)
```

Condition for Movement:

```
self.threshold=0.007 (line 115)
if np.linalg.norm(self.e) < self.threshold '(line 131)
```

This condition check is a termination criterion for the iterative visual servoing process. Once the Euclidean norm of the error vector falls below the predefined threshold, the code assumes that the robot has reached the desired target position and orientation, and it disables further movement and shuts down MoveIt resources.

The Euclidean norm of the error vector is a measure of the magnitude or distance of the error in the visual servoing system. When this norm falls below a predefined threshold, it indicates that the current error or difference between the actual and target positions is small enough, and the system can be considered to have reached a satisfactory or accurate position.

target_callback

Is almost the same with version one, it react to service request instead.

Launch the Gazebo ans Run the ROS nodes and observe the robot's movement in response to the Aruco tag.

```
roslaunch m0609_moveit_config dsr_moveit_gazebo.launch
roslaunch visual_servoing vsm.launch
```

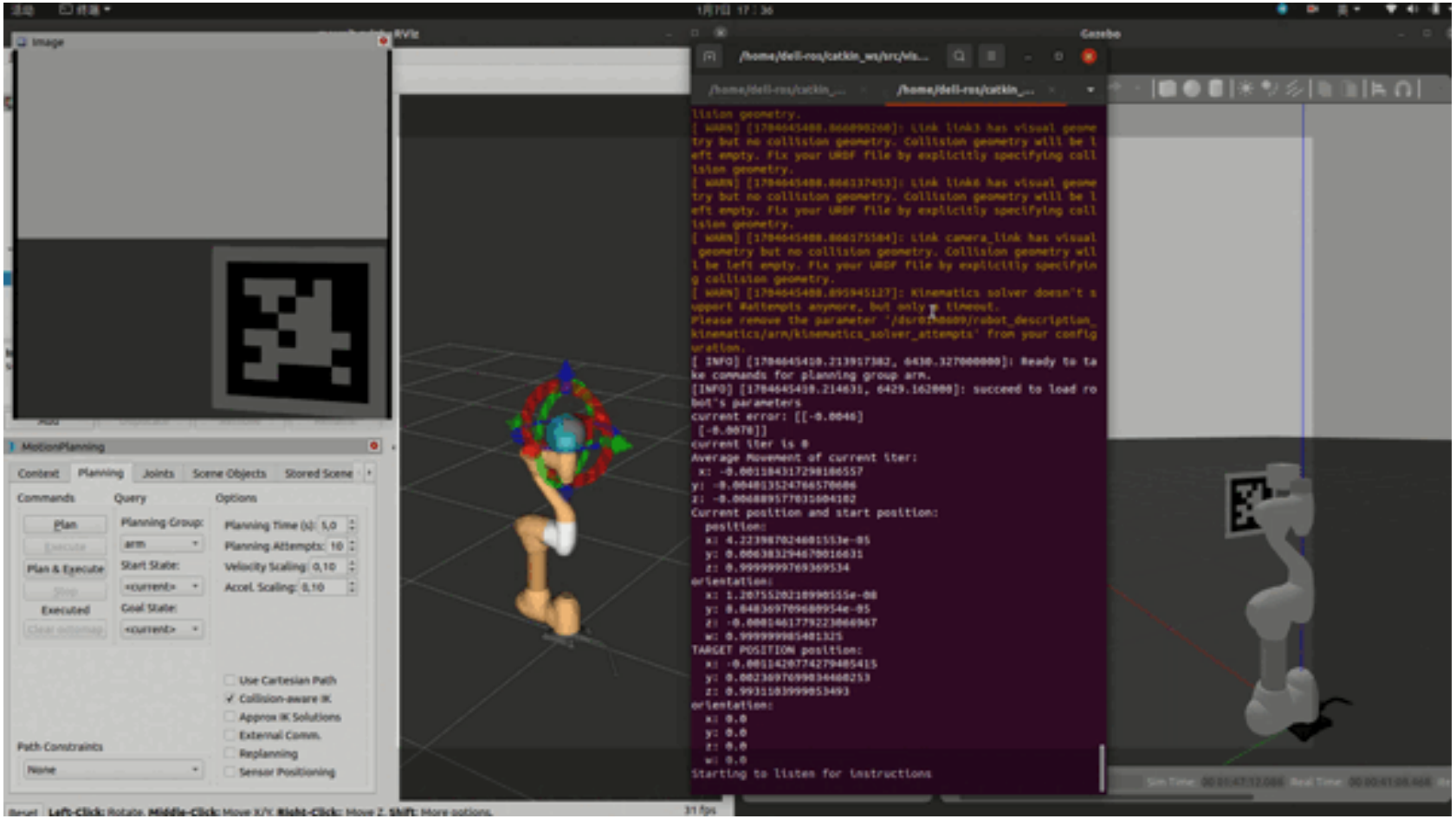


Image
RViz
Gazebo
/home/dell-ros/catkin_ws/src/vis...
/home/dell-ros/catkin_...
/home/dell-ros/catkin_...
lision geometry.
[WARN] [1704645408.866090260]: Link link3 has visual geometry but no collision geometry. Collision geometry will be left empty. Fix your URDF file by explicitly specifying collision geometry.
[WARN] [1704645408.866137453]: Link link4 has visual geometry but no collision geometry. Collision geometry will be left empty. Fix your URDF file by explicitly specifying collision geometry.
[WARN] [1704645408.866175104]: Link camera_link has visual geometry but no collision geometry. Collision geometry will be left empty. Fix your URDF file by explicitly specifying collision geometry.
[WARN] [1704645408.895945127]: Kinematics solver doesn't support #attempts anymore, but only a timeout.
Please remove the parameter '/dsr01m0609/robot_description_kinematics/arm/kinematics_solver_attempts' from your configuration.
[INFO] [1704645410.213917382, 6430.327000000]: Ready to take commands for planning group arm.
[INFO] [1704645410.214631, 6429.162000]: succeed to load robot's parameters
current error: [[-0.0046]
 [-0.0078]]
current iter is 0
Average Movement of current iter:
 x: -0.0011843172981865 57
y: -0.0040135247665706 86
z: -0.0068895770316041 02
Current position and start position:
  position:
    x: 4.223987824681553e-05
    y: 0.0003832944670016031
    z: 0.9999999769369534
  orientation:
    x: 1.2075520218990555e-08
    y: 0.0048369709680954e-05
    z: -0.0001461779223866967
    w: 0.9999999685481325
TARGET POSITION position:
    x: -0.0011420774279485415
    y: 0.0023697699834668253
    z: 0.9931103999853493
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0
Starting to listen for instructions
MotionPlanning
Context
Planning
Joints
Scene Objects
Stored Scene
Commands
Query
Options
Plan
Execute
Plan & Execute
Stop
Executed
Clear octomap
Planning Group:
arm
Start State:
<current>
Goal State:
<current>
Planning Time (s): 5,0
Planning Attempts: 10
Velocity Scaling: 0,10
Accel. Scaling: 0,10
Use Cartesian Path
Collision-aware IK
Approx IK Solutions
External Comm.
Replanning
Sensor Positioning
Path Constraints
None
Reset
Left-Click: Rotate, Middle-Click: Move X/Y, Right-Click: Move Z, Shift: More options.
31 fps