Robotic Arm : Grasping task: Documentation

File: demo.py

This file comprises of code which performs the data collection by running grasping simulations on the loaded object urdf.

Function name: move_arm_and_rotate :

This function would set the robotic arm at a random angle before initiating to approach the object

Function name: approach_object:

This function initiates the robotic arm to move towards the object

Function name: return_object_to_original_robot_pose:

This function allows the robotic arm to come back to its original position after attempting to grasp the underlying object.

Function name: reset_object:

This function resets the object to grasp for successive simulations

The above functions description can be known by referring https://www.etedal.net/2020/04/pybullet-panda_2.html

Function name: get_grasp_img:

This function facilitates in fetching the snapshots at various instances during the grasp.

Function name: grip_object:

This function executes the robotic arm fingers to inch towards grasping the object.
During this function's execution, we ought to record
-> Pre-grasp robot coordinates
-> Post grasp robot coordinates
-> Robotic arm left finger's point cloud info (the information on contact points of robot's left finger)
-> Robotic arm right finger's point cloud info (the information on contact points of robot's right finger)

Each of these functions involve using inbuilt pybullet functions which can be referred from :
http://dirkmittler.homeip.net/blend4web_ce/uranium/bullet/docs/pybullet_quickstartguide.pdf

Pre/Post grasp robot coordinates:

During capturing of robotic coordinates, we try to capture all the joint level information.

The robot under simulation is having 11 joints (numbered from 0 to 10)

For each of the joint, we fetch the joint information by calling the getjointstate(robot_urdf, joint_number) function.

The get joint state function returns the below information:

getJointState output

jointPosition	float	The position value of this joint.
jointVelocity	float	The velocity value of this joint.
jointReactionForces	list of 6 floats	There are the joint reaction forces, if a torque sensor is enabled for this joint. Without torque sensor, it is [0,0,0,0,0,0].
appliedJointMotorTorque	float	This is the motor torque applied during the last stepSimulation

We would need to fetch all these information and store it individually and thus
If joint_info = getjointstate(robot_urdf, joint_number)

Then:
Joint_info[0] -> joint position
Joint_info[1] -> joint velocity
Joint_info[2] -> joint reaction forces

Joint_info[3] -> motor torque

In addition to the joint level information, we capture the link level information of the link between the two joint fingers (joint fingers are joint 9 and joint 10, and the link between them is link 11)

We use the getlinkstate function(robot_urdf,link_number) function which returns information as shown below:

getLinkState return values

linkWorldPosition	vec3, list of 3 floats	Cartesian position of center of mass
linkWorldOrientation	vec4, list of 4 floats	Cartesian orientation of center of mass, in quaternion [x,y,z,w]
localInertialFramePosition	vec3, list of 3 floats	local position offset of inertial frame (center of mass) to URDF link frame
localInertialFrameOrientation	vec4, list of 4 floats	local orientation (quaternion [x,y,z,w]) offset of the inertial frame to the URDF link frame.

Out of these only the world position is our interested value, and thus we fetch getlinkstate(robot_urdf,joint_number)[0]

All these details totally in combination sum up to a robotic coordinate information, and we thus capture these values during the pre-grasp and post-grasp points.

Robotic arm point cloud info [left/right]

To capture the point cloud (the points of contact of the left and right gripper of the robot while trying to grasp an object), we leverage the getcontactpoints function which returns values as shown:

getContactPoints will return a list of contact points. Each contact point has the following fields:

contactFlag	int	reserved
bodyUniqueIdA	int	body unique id of body A
bodyUniqueIdA	int	body unique id of body B
linkIndexA	int	link index of body A, -1 for base
linkIndexB	int	link index of body B, -1 for base
positionOnA	vec3, list of 3 floats	contact position on A, in Cartesian world coordinates
positionOnB	vec3, list of 3 floats	contact position on B, in Cartesian world coordinates
contactNormalOnB	vec3, list of 3 floats	contact normal on B, pointing towards A
contactDistance	float	contact distance, positive for separation, negative for penetration
normalForce	float	normal force applied during the last 'stepSimulation'

If contact_information = getcontactpoints(object_urdf,robot_urdf)

Out of these the information of interest include:

positionOnA : obtained by contact_information[5]
positionOnB : obtained by contact_information[6]
Contact Normal: obtained by contact_information[7]
Contact distance obtained by contact_information[8]
Normal force : obtained by contact_information[9]

We try to see if the information belongs to joint 9 or joint 10 through the “linkIndexB” parameter (contact_information[4])

Additionally we ought to ensure that the contact information is an actual information. To filter the actual information out, we ensure the contact distance (contact_information[8]) is in the range of 1e-3.

To run the code:

python -u demo.py --object <object_name> --num_sim <no.of simulations to run> --folder_path <path where the json results need to be stored>

Eg:

python -u demo.py --object soap --num_sim 5 --folder_path grasp_soap

To view the json in a more readable format, if soap_data.json is the json output, run this command to turn the json into a readable json file

cat output_data/soap_data.json | python -mjson.tool > output_data/aligned_soap_data.json

The aligned output is stored in aligned_soap_data.json