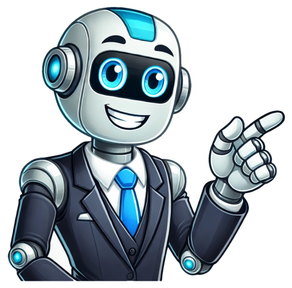
Continue



You cant perform that action at this time. MuJoCo ++ pyroboplan startgoal q_start = self.random_valid_state() q_goal = self.random_valid_state() RRT planner = RRTPlanner(model, collision_model, options=options) optimizer = CubicTrajectoryOptimization(model, collision_model, options) matplotlib # positions = []for tform in tforms: position = tform.translation positions.append(position)positions = np.array(positions)# 3D fig = plt.figure()ax = fig.add_subplot(111, projection='3d')# ax.plot(positions[:, 0], positions[:, 1], positions[:, 2], marker='o')# for i, tform in enumerate(tforms): position = tform.translation rotation_matrix = tform.rotation # x_axis = rotation_matrix[:, 0] y_axis = rotation_matrix[:, 1] z_axis = rotation_matrix[:, 2] # ax.quiver(position[0], position[1], position[2], x_axis[0], x_axis[1], x_axis[2], color='r', length=0.1) ax.quiver(position[0], position[1], position[2], y_axis[0], y_axis[1], y_axis[2], color='g', length=0.1) ax.quiver(position[0], position[1], position[2], z_axis[0], z_axis[1], z_axis[2], color='b', length=0.1)# ax.set_xlabel('X')ax.set_ylabel('Y')ax.set_zlabel('Z')# plt.show(block=False)plt.pause(0.001) mujoco viewer import mujoco_viewerimport mujoco,time,threadingimport numpy as npimport pinocchioimport matplotlib# matplotlib.use('TkAgg')import matplotlib.pyplot as pltfrom mpl_toolkits.mplot3d import Axes3Dfrom pyroboplan.core.utils import (get_random_collision_free_state, extract_cartesian_poses,)from pyroboplan.models.panda import (load_models, add_self_collisions, add_object_collisions,)from pyroboplan.planning.rrt import RRTPlanner, RRTPlannerOptionsfrom pyroboplan.trajectory.trajectory_optimization import (CubicTrajectoryOptimization, CubicTrajectoryOptimizationOptions,)class Test(mujoco_viewer.CustomViewer): def __init__(self, path): super().__init__(path, 3, azimuth=180, elevation=-30) self.path = path def runBefore(self): # Create models and data self.model_roboplan, self.collision_model, visual_model = load_models(use_sphere_collisions=True) add_self_collisions(self.model_roboplan, self.collision_model) add_object_collisions(self.model_roboplan, self.collision_model, visual_model, inflation_radius=0.1) data = self.model_roboplan.createData() collision_data = self.collision_model.createData() self.target_frame = "panda_hand" ignore_joint_indices = [self.model_roboplan.getJointId("panda_finger_joint1") - 1, self.model_roboplan.getJointId("panda_finger_joint2") - 1,] np.set_printoptions(precision=3) self.distance_padding = 0.001 self.init_state = self.data.qpos.copy() while True: q_start = self.random_valid_state() q_goal = self.random_valid_state() # Search for a path options = RRTPlannerOptions(max_step_size=0.05, max_connection_dist=5.0, rrt_connect=False, bidirectional_rrt=True, rrt_star=True, max_rewire_dist=5.0, max_planning_time=20.0, fast_return=True, goal_biasing_probability=0.15, collision_distance_padding=0.01,) print("") print(f"Planning a path...") planner = RRTPlanner(self.model_roboplan, self.collision_model, options=options) q_path = planner.plan(q_start, q_goal) if len(q_path) > 0: print(f"Got a path with {len(q_path)} waypoints") else: print("Failed to plan.") # Perform trajectory optimization. dt = 0.025 options = CubicTrajectoryOptimizationOptions(num_waypoints=len(q_path), samples_per_segment=7, min_segment_time=0.5, max_segment_time=10.0, min_vel=-1.5, max_vel=1.5, min_accel=-0.75, max_accel=0.75, min_jerk=-1.0, max_jerk=1.0, max_planning_time=30.0, check_collisions=True, min_collision_dist=self.distance_padding, collision_influence_dist=0.05, collision_avoidance_cost_weight=0.0, collision_link_list=["obstacle_box_1", "obstacle_box_2", "obstacle_sphere_1", "obstacle_sphere_2", "ground_plane", "panda_hand",],) print("Optimizing the path...") optimizer = CubicTrajectoryOptimization(self.model_roboplan, self.collision_model, options) traj = optimizer.plan([q_path[0], q_path[-1]], init_path=q_path) if traj is None: print("Retrying with all the RRT waypoints...") traj = optimizer.plan(q_path, init_path=q_path) if traj is not None: print("Trajectory optimization successful") traj_gen = traj.generate(dt) self.q_vec = traj_gen[1] print(f"path has {self.q_vec.shape[1]} points") tforms = extract_cartesian_poses(self.model_roboplan, "panda_hand", self.q_vec.T) # positions = [] for tform in tforms: position = tform.translation positions.append(position) positions = np.array(positions) # 3D fig = plt.figure() ax = fig.add_subplot(111, projection='3d') # ax.plot(positions[:, 0], positions[:, 1], positions[:, 2], marker='o') # for i, tform in enumerate(tforms): position = tform.translation rotation_matrix = tform.rotation # x_axis = rotation_matrix[:, 0] y_axis = rotation_matrix[:, 1] z_axis = rotation_matrix[:, 2] # ax.quiver(position[0], position[1], position[2], x_axis[0], x_axis[1], x_axis[2], color='r', length=0.01) ax.quiver(position[0], position[1], position[2], y_axis[0], y_axis[1], y_axis[2], color='g', length=0.01) ax.quiver(position[0], position[1], position[2], z_axis[0], z_axis[1], z_axis[2], color='b', length=0.01) # ax.set_xlabel('X') ax.set_ylabel('Y') ax.set_zlabel('Z') # plt.show(block=False) plt.pause(0.001) break self.index = 0 def random_valid_state(self): return get_random_collision_free_state(self.model_roboplan, self.collision_model, distance_padding=0.01) def runFunc(self): self.data.qpos[:7] = self.q_vec[:7, self.index] self.index += 1 if self.index >= self.q_vec.shape[1]: self.index = 0 time.sleep(0.01)if __name__ == "__main__": test = Test("/home/dar/MuJoCoBin/mujoco_menagerie/franka_emika_panda/scene.xml") test.run_loop() pyroboplan Motion planning first requires a model of the robot.This model must describe:Robot kinematics (and possibly its dynamics).Geometries of the robot and its environment for visualization and collision checking.Names of joints and/or coordinate frames that are used for motion planning.Support for importing standard robot description file formats such as URDF.One software package that has all these capabilities is Pinocchio.It is actively maintained and therefore open to contributions, and is performant because it is written in C++ with first-class Python bindings.This applies to both Pinocchio itself, and the Coal library used internally for collision checking.For completeness, other good alternatives for robot modeling would be Drake, MuJoCo, and Robotics Toolbox for Python.Pinocchio offers several methods of visualization; in this package, we use MeshCat. While pyroboplan extensively relies on other tools (mostly Pinocchio) for robot modeling, it implements several components of motion planning from scratch.The goal of this library is learning and education, and as such its design philosophy is as follows:Motion planning components (inverse kinematics, motion planning, trajectory generation, etc.) must be easy to mix and match.There is no need for unnecessarily fancy software techniques like factories or excessive inheritance. Its OK for different implementations to have slightly different interfaces if it makes sense.The math must always be explained, and resources to relevant papers, lectures, tutorials, etc. should be cited wherever possible.Visualization, debugging, and clarity of code is key and should be prioritized in favor of efficiency/performance.Of course, motion planning is a vast field and this library will never have all the implementations available.This is where we hope that you will be inspired by this project to do some learning yourself, implement new capabilities, and share your contributions with the community.If you are looking for motion planning libraries that are less education-focused and more capable, some of the standard ones used in practice are:Drake: Contains tools for system modeling and optimization-based planning and control.MoveIt: A ROS based framework with a plugin-based system for inverse kinematics, motion planning, and pre-/post-processing. Best known for its interface to Open Motion Planning Library (OMPL) for sampling-based motion planning.Tesseract: A library inspired by MoveIt concepts, but written to better decouple the code functionality and ROS interfaces, and with a bigger focus on trajectory optimization.OCS2 and Crocoddyl: Packages based on Pinocchio for optimization-based online control. Even though the implementations of motion planning algorithms themselves arent held to a professional software engineering standard, in favor of educational goals, all the supporting infrastructure is still (somewhat) professional.Versioning: This library will be versioned using the typical SemVer specification and periodically released.Documentation: Ensure that all of your modules, top-level files, and functions have docstrings, as they get autogenerated into this doc page. You can look at existing files for inspiration, and generate the docs locally to check how things look.Unit testing: Testing is awesome at ensuring your code works as intended and doesnt break other things. You should implement tests for whatever you write! Copyright 2024-2025, Sebastian Castro. Built with Sphinx using a theme provided by Read the Docs. JavaScript is disabled in your browser. Please enable JavaScript to proceed. Copyright 2024-2025, Sebastian Castro. Built with Sphinx using a theme provided by Read the Docs. PyRoboPlan PythonmanipulatorPinocchioPython PyRoboPlan PyRoboPlan PyPI PyPI pip3 install pyroboplan : git clone source scripts/setup_virtual_env.shpip3 install -e . PyRoboPlan from pyroboplan.robot_modeling import create_robot_modelfrom pyroboplan.motion_planning import inverse_kinematicsrobot = create_robot_model('example_robot') # 'example_robot'target_pose = [x, y, z, roll, pitch, yaw] # joint_angles = inverse_kinematics(robot, target_pose)print(":", joint_angles) PyRoboPlanPinocchioPyRoboPlan PyRoboPlanGitHub