mplib

Release 0.2.0a0

Minghua Liu, Jiayuan Gu, Kolin Guo, Xinsong Lin

Apr 16, 2024

CONTENTS

1 Installation 3

2 Usage 5

3 Examples 7

4 API Reference 21

Python Module Index 103

Index 105

MPlib is a lightweight python package for motion planning, which is decoupled from ROS and is easy to set up. With a few lines of python code, one can achieve most of the motion planning functionalities in robot manipulation.

CHAPTER

ONE

INSTALLATION

Pre-built pip packages support Ubuntu 20.04+ with Python 3.8+.

```
pip install mplib
```

CHAPTER

TWO

USAGE

See our tutorial for detailed usage and examples.

CHAPTER

THREE

EXAMPLES

3.1 Motion Planning Examples

3.1.1 Getting Started

Installation

`mplib` is a lightweight python package that includes common functionalities for motion planning. You can use `mplib` to plan a collision-free trajectory for a robot, calculate inverse kinematics, and take point cloud observation as an environment model. Unlike MoveIt, `mplib` is decoupled from ROS, and it's easy to set up and use with simple python APIs.

Please use pip to install `mplib`:

```
pip install mplib
```

Supported Python versions: 3.6+

Supported operating system: Ubuntu 18.04+

Planner Configuration

To use `mplib`, we need to first set up a planner for the robot with the following constructor:

```
    def __init__(
        self,
        urdf: str | Path,
        move_group: str,
        *,
        srdf: Optional[str | Path] = None,
        new_package_keyword: str = "",
        use_convex: bool = False,
        user_link_names: Sequence[str] = [],
        user_joint_names: Sequence[str] = [],
        joint_vel_limits: Optional[Sequence[float] | np.ndarray] = None,
        joint_acc_limits: Optional[Sequence[float] | np.ndarray] = None,
        objects: list[FCLObject] = [],  # noqa: B006
        verbose: bool = False,
    ):
```

- The URDF file describes the robot, while the SRDF file complements the URDF and specifies additional information for motion planning. For example, `mplib` loads the `disable_collisions` pairs in SRDF to ignore collisions between certain pairs of links such as adjacent joints. Currently, SRDF files are generated by `MoveIt Setup Assistant`. In the future, we may provide other tools for generating SRDF files.
- To specify the link order and joint order, one can provide `user_link_names` and `user_joint_names`. By default, the joints and links are in the order they are loaded. However, if you are using a simulation library such as `sapien`, you might need to change this. Please note that we only care about the active joints (i.e., revolute and prismatic joints) and ignore the fixed joints.
- `move_group` specifies the target link[1] for which we may specify target poses to reach. The end-effector of an agent is typically specified as the `move_group`. After specifying the `move_group`, `mplib` only focuses on those active joints along the path from the root link to the `move_group`, since other joints doesn't affect the pose of the `move_group`. For example, for our `panda` robot arm (7 DoF), the end-effector is `panda_hand`. Only the first seven active joints affect the pose of `panda_hand`, while the last two finger joints don't.
- For safety, the robot cannot move arbitrarily fast. `joint_vel_limits` and `joint_acc_limits` specify the maximum joint velocity and maximum joint acceleration constraints for the active joints along the path from the root link to the `move_group`. `mplib` takes the constraints into account when solving the time-optimal path parameterization. By default, `mplib` uses 1.0m/s and 1.0m/s^2 as the default joint velocity and acceleration limits.

After setting up the planner, we can use it to solve many motion planning tasks.

3.1.2 Plan a Path

In this tutorial, we will talk about how to plan paths for the agent. As shown in the demo, the robot needs to move the three boxes a bit forward. The full script can be found here `demo.py`. You will also need `demo_setup.py` and grab the panda URDF.

Fig. 1: plan with RRTConnect

Note: This tutorial only talks about the basic usages, and the robot only avoids self-collisions (i.e., collisions between the robot links) in this demo. Please refer to *Collision Avoidance* to include the environment model and other advanced usages.

Plan with sampling-based algorithms

`mplib` supports state-of-the-art sampling-based motion planning algorithms by leveraging OMPL. You can call `planner.plan_pose()` to plan a path for moving the `move_group` link to a target pose:

```
        print("plan_pose")
        result = self.planner.plan_pose(pose, self.robot.get_qpos(), time_step=1 / 250)
```

Specifically, `planner.plan_pose()` takes two required arguments as input. The first one is the target pose of the `move_group` link. It's a 7-dim list, where the first three elements describe the position part, and the remaining four elements describe the quaternion (wxyz) for the rotation part. **Note that the pose is relative to the world frame**. Normally, the base link of the robot is the world frame unless you have called `set_base_pose(new_pose)` in on the planner. You can also temporarily plan w.r.t. the robot base by passing in `wrt_world=False`.

[1] `mplib` currently only supports a single link as `move_group`.

The second argument is the current joint positions of all the active joints (not just all the active joints in the movegroup). The `planner.plan_pose()` function first solves the inverse kinematics to get the joint positions for the target pose. It then calls the RRTConnect algorithm to find a path in the joint space. Finally, it simplifies the path and parameterizes the path to generate time, velocity, and acceleration information.

`planner.plan_pose()` returns a dict which includes:

- `status`: a string indicates the status:
  - `Success`: planned a path successfully.
  - `IK Failed`: failed to solve the inverse kinematics. This may happen when the target pose is not reachable.
  - `RRT Failed`: failed to find a valid path in the joint space. This may happen when there is no valid path or the task is too complicated.
- `position`: a NumPy array of shape $(n \times m)$ describes the joint positions of the waypoints. n is the number of waypoints in the path, and each row describes a waypoint. m is the number of active joints that affect the pose of the `move_group` link. For example, for our panda robot arm, each row includes the positions for the first seven joints.
- `duration`: a scalar indicates the duration of the output path. `mplib` returns the optimal duration considering the velocity and acceleration constraints.
- `time`: a NumPy array of shape (n) describes the time step of each waypoint. The first element is equal to 0, and the last one is equal to the `duration`. Argument `time_step` determines the interval of the elements.
- `velocity`: a NumPy array of shape $(n \times m)$ describes the joint velocities of the waypoints.
- `acceleration`: a NumPy array of shape $(n \times m)$ describing the joint accelerations of the waypoints.

`planner.plan_pose()` also takes other optional arguments with default values:

Planner.**plan_pose**(
goal_pose: Pose,
current_qpos: ndarray,
mask: list[bool] | ndarray | None = None,
*,
time_step: float = 0.1,
rrt_range: float = 0.1,
planning_time: float = 1,
fix_joint_limits: bool = True,
wrt_world: bool = True,
simplify: bool = True,
constraint_function: Callable | None = None,
constraint_jacobian: Callable | None = None,
constraint_tolerance: float = 1e-3,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

plan from a start configuration to a goal pose of the end-effector

Parameters

- **goal_pose** – pose of the goal
- **current_qpos** – current joint configuration (either full or move_group joints)
- **mask** – if the value at a given index is True, the joint is *not* used in the IK
- **time_step** – time step for TOPPRA (time parameterization of path)
- **rrt_range** – step size for RRT

- **planning_time** – time limit for RRT
- **fix_joint_limits** – if True, will clip the joint configuration to be within the joint limits
- **wrt_world** – if true, interpret the target pose with respect to the world frame instead of the base frame
- **verbose** – if True, will print the log of OMPL and TOPPRA

Follow a path

`plan_pose()` outputs a time-parameterized path, and we need to drive the robot to follow the path. In this demo, we use `sapien` to simulate and drive the robot.

```
    def follow_path(self, result):
        """Helper function to follow a path generated by the planner"""
        # number of waypoints in the path
        n_step = result["position"].shape[0]
        # this makes sure the robot stays neutrally boyant instead of sagging
        # under gravity
        for i in range(n_step):
            qf = self.robot.compute_passive_force(
                gravity=True, coriolis_and_centrifugal=True
            )
            self.robot.set_qf(qf)
            # set the joint positions and velocities for move group joints only.
            # The others are not the responsibility of the planner
            for j in range(len(self.planner.move_group_joint_indices)):
                self.active_joints[j].set_drive_target(result["position"][i][j])
                self.active_joints[j].set_drive_velocity_target(
                    result["velocity"][i][j]
                )
            # simulation step
            self.scene.step()
            # render every 4 simulation steps to make it faster
            if i % 4 == 0:
                self.scene.update_render()
                self.viewer.render()
```

Note: If you find your robot doesn't move as expected, please **double-check** your controller, especially the controller's parameters. In many cases, the planner finds a good path while the controller fails to follow the path.

Plan with screw motion

Besides using the sampling-based algorithms, we also provide another simple way (trick) to plan a path. For some tasks, we can directly move the `move_group` link towards the target pose. It's internally achieved by first calculating the relative transformation from its current pose to the target pose, then calculating the relative transformation's exponential coordinates, and finally calculating the joint velocities with the Jacobian matrix.

Compared to the sampling-based algorithms, planning with screw motion has the following pros:

- faster: since it doesn't need to sample lots of states in the joint space, planning with screw motion can save lots of planning time.
- *straighter* path: there is no guarantee for sampling-based algorithms to generate *straight* paths even it's a simple lifting task since it connects states in the joint space. In contrast, the returned path by the exponential coordinates and the Jacobian matrix can sometimes be more reasonable. See the above figures for comparison.

You can call `planner.plan_screw()` to plan a path with screw motion. Similar to `planner.plan_pose()`, it also takes two required arguments: target pose and current joint positions, and returns a dict containing the same set of elements.

```
    def plan_screw(
        self,
        goal_pose: Pose,
        current_qpos: np.ndarray,
        *,
        qpos_step: float = 0.1,
        time_step: float = 0.1,
        wrt_world: bool = True,
        verbose: bool = False,
    ) -> dict[str, str | np.ndarray | np.float64]:
```

However, planning with screw motion only succeeds when there is no collision during the planning since it can not detour or replan. We thus recommend use `planner.plan_screw()` for some simple tasks or combined with `planner.plan_pose()`. As shown in the code, we first try `planner.plan_screw()`, if it fails (e.g., collision during the planning), we then turn to the sampling-based algorithms. Other arguments are the same with `planner.plan_pose()`.

Move the boxes

In this example, we create some boxes inside the simulation like so:

```
        builder = self.scene.create_actor_builder()
        builder.add_box_collision(half_size=[0.02, 0.02, 0.06])
        builder.add_box_visual(half_size=[0.02, 0.02, 0.06])
        red_cube = builder.build(name="red_cube")
        red_cube.set_pose(sapien.Pose([0.4, 0.3, 0.06]))

        builder = self.scene.create_actor_builder()
        builder.add_box_collision(half_size=[0.02, 0.02, 0.04])
        builder.add_box_visual(half_size=[0.02, 0.02, 0.04])
        green_cube = builder.build(name="green_cube")
        green_cube.set_pose(sapien.Pose([0.2, -0.3, 0.04]))

        builder = self.scene.create_actor_builder()
        builder.add_box_collision(half_size=[0.02, 0.02, 0.07])
        builder.add_box_visual(half_size=[0.02, 0.02, 0.07])
```

(continues on next page)

(continued from previous page)

```
        blue_cube = builder.build(name="blue_cube")
        blue_cube.set_pose(sapien.Pose([0.6, 0.1, 0.07]))
```

We then find the target poses needed to reach the boxes.

```
        poses = [
            [0.4, 0.3, 0.12, 0, 1, 0, 0],
            [0.2, -0.3, 0.08, 0, 1, 0, 0],
            [0.6, 0.1, 0.14, 0, 1, 0, 0],
        ]
```

Then, we plan and execute the motion:

```
        for i in range(3):
            pose = poses[i]
            pose[2] += 0.2
            self.move_to_pose(pose)
            self.open_gripper()
            pose[2] -= 0.12
            self.move_to_pose(pose)
            self.close_gripper()
            pose[2] += 0.12
            self.move_to_pose(pose)
            pose[0] += 0.1
            self.move_to_pose(pose)
            pose[2] -= 0.12
            self.move_to_pose(pose)
            self.open_gripper()
            pose[2] += 0.12
            self.move_to_pose(pose)
```

Fig. 2: plan with screw motion

3.1.3 Inverse Kinematics

Inverse kinematics determine the joint positions that provide the desired pose for the robot's end-effectors. In `mplib`, you can solve the inverse kinematics of the `move_group` link with:

`Planner.IK(`
goal_pose: Pose,
start_qpos: ndarray,
mask: Sequence[bool] | ndarray | None = None,
***,
n_init_qpos: int = 20,
threshold: float = 1e-3,
return_closest: bool = False,
verbose: bool = False,
`) →` tuple[str, list[ndarray] | ndarray | None]

Compute inverse kinematics

Parameters

- **goal_pose** – goal pose
- **start_qpos** – initial configuration, (ndof,) np.floating np.ndarray.
- **mask** – qpos mask to disable IK sampling, (ndof,) bool np.ndarray.
- **n_init_qpos** – number of random initial configurations to sample.
- **threshold** – distance threshold for marking sampled IK as success. distance is position error norm + quaternion error norm.
- **return_closest** – whether to return the qpos that is closest to start_qpos, considering equivalent joint values.
- **verbose** – whether to print collision info if any collision exists.

Returns

(status, q_goals)

status: IK status, "Success" if succeeded.

q_goals: list of sampled IK qpos, (ndof,) np.floating np.ndarray.
IK is successful if q_goals is not None. If return_closest, q_goals is np.ndarray if successful and None if not successful.

`Planner.IK()` internally implements a numerical method and takes the following arguments:

- `target_pose`: a 7-dim list specifies the target pose of the `move_group` link. The first three elements describe the position part, and the remaining four elements describe the quaternion (wxyz) for the rotation part.
- `init_qpos`: a list describes the joint positions of all the active joints (e.g., given by SAPIEN). It will be used as the initial state for the numerical method.
- `mask`: a list of 0/1 values with the same length as `init_qpos`. It specifies which joints are disabled (1). For example, if you want to solve the inverse kinematics of the first 2 joints, you can set `mask=[0,0,1,1,1,1,1]`.
- `n_init_qpos=20`: besides the provided initial state, the method also samples extra initial states to run the algorithm for at most `n_init_qpos` times. In this way, it can avoid local minimums and increase the success rate.
- `threshold=1e-3`: a threshold for determining whether the calculated pose is close enough to to the target pose.

It returns a tuple of two elements:

- `status`: a string indicates the status.
- `result`: a NumPy array describes the calculated joint positions.

Note: If `planner.IK()` fails, please increase `n_init_qpos` or double-check whether the target pose is reachable.

3.1.4 Collision Avoidance

In *Plan a Path*, we talked about how to plan paths for the robot. However, in that tutorial, we didn't take the environment model into account. The robot will avoid self-collisions (i.e., collisions between the robot links), but may collide with the environment.

This tutorial will introduce two ways two ways to avoid collisions: add environment point clouds and attach a box. As shown in the right figure, the robot needs to move the red box to the place of the green box while avoiding collision with the blue box. The full script can be found here `demo.py`. You will also need `demo_setup.py` and grab the panda URDF..

The above figures show the benefit of collision avoidance:

- Left: w/o point cloud, w/o attach. The robot arm hits the blue box.
- Middle: w/ point cloud, w/o attach. The red box hits the blue box.
- Right: w/ point cloud, w/ attach. There is no collision.

Add environment point clouds

One way to model the environment and avoid collision is through point clouds. The point cloud may come from the sensor observations or be sampled from the mesh surfaces. For example, we can add a point cloud for the blue box with `planner.update_point_cloud()`:

```
        box = trimesh.creation.box([0.1, 0.4, 0.2])
        points, _ = trimesh.sample.sample_surface(box, 1000)
        points += [0.55, 0, 0.1]
        self.planner.update_point_cloud(points, resolution=0.02)
```

`planner.update_point_cloud()` takes two arguments. The first one is a NumPy array of shape $(n \times 3)$, which describes the coordinates of the points. **The coordinates should be represented in the world frame**. The second (optional) argument is `resolution`, which describes the resolution of each point. This can be used to create a buffer around the collision object.

You don't need to provide the point cloud for each `planner.plan()` or `planner.plan_screw()` call. You can use `planner.update_point_cloud()` to update the point cloud once it's changed.

Note: Please remember to remove the points of the robot arm if the points come from the sensor observation. Otherwise, there will always be collisions, and the planner may fail to plan a valid path.

Attach a box

As shown in the above figure (middle one), after adding the point cloud of the blue box, the robot will not collide with it. However, the red box moves with the robot, and it may still collide with the blue box. To address this issue, we can attach a box to the robot, so that we can avoid the collision between the attached box and the environment point cloud:

```
        if use_attach:
            self.planner.update_attached_box(
                [0.04, 0.04, 0.12], [0, 0, 0.14, 1, 0, 0, 0]
            )
```

`planner.update_attached_box()` takes three arguments:

- `size`: a list with three elements indicates the size of the attached box.
- `pose`: a list with seven elements indicates the relative pose from the box to the attached link. The first three elements describe the position part, and the remaining four elements describe the quaternion (wxyz) for the rotation part.
- `link_id = -1`: optional, an integer indicates the id of the link that the box is attached to. The link id is determined by the `user_link_names` (during Configuration), and starts from 0. The default value -1 indicates the `move_group` link.

After adding the attached box, we can avoid collisions between the attached box and the point cloud by setting both `use_point_cloud` and `use_attach` to be True. Both `planner.plan_pose()` and `planner.plan_screw()` support the flags.

You can use `planner.update_attached_box()` again to update the box once it's changed.

As shown in the above figure (the right one), after adding the point cloud of the blue box and attaching the red box to the `move_group` link, there is no collision.

Note: There are also a family of update attach functions. One can attach a sphere or a general mesh. Please see the planner API for more details.

3.1.5 Detecting Collision

In this tutorial, we will see how to use the planner to detect collisions without planning a path. There are two APIs that are wrappers around some *fcl* library functions to provide a more convenient interface. In particular, we have *check_for_self_collision* and *check_for_env_collision*. As the name suggests, the former checks for robot self-collision, while the latter checks for collision between the robot and its environment.

Setting Up the Planner

We will use the convenient function *setup_planner* provided by *mplib.examples.demo_setup.DemoSetup* to load the robot and create a planner. We will also make a function to print out the collisions detected.

```
    def print_collisions(self, collisions):
        """Helper function to abstract away the printing of collisions"""
        if len(collisions) == 0:
            print("No collision")
            return
        for collision in collisions:
            print(
                f"{collision.link_name1} of entity {collision.object_name1} collides"
                f" with {collision.link_name2} of entity {collision.object_name2}"
            )
```

We will also create a floor as one of the collision objects to demonstrate the *check_for_env_collision* API.

```
        floor = fcl.Box([2, 2, 0.1])  # create a 2 x 2 x 0.1m box
        # create a collision object for the floor, with a 10cm offset in the z direction
        floor_fcl_collision_object = fcl.CollisionObject(floor, Pose(p=[0, 0, -0.1]))
        # update the planning world with the floor collision object
        self.planner.planning_world.add_object("floor", floor_fcl_collision_object)
```

Note that we call floor an object because it is not an articulated object. The function to add an object to the planning world is *add_object*. This can also be used to update the pose of the object or change it our entirely.

Collision Time!

We will now test several configurations to see how the planner detects collisions. First, we will set the robot to a self-collision-free qpos and check for self-collision. This should return no collision. Note that the full joint configuration is not provided here. Instead, on the movegroup related joints are set. The rest of the joints are set to the current joint angle.

```
        # if the joint qpos does not include the gripper joints,
        # it will be set to the current gripper joint angle
        self_collision_free_qpos = [0, 0.19, 0.0, -2.61, 0.0, 2.94, 0.78]
        self.print_collisions(
            self.planner.check_for_self_collision(self_collision_free_qpos)
        )
```

Next, we will put the robot into a self-collision qpos and check for self-collision. This should return a collision.

```
        self_collision_qpos = [0, 1.36, 0, -3, -3, 3, -1]
        self.print_collisions(
            self.planner.check_for_self_collision(self_collision_qpos)
        )
```

Then, we do the same thing with environment collision as we put the robot into a pose that collides with the floor. Additionally, we also try to plan a path to this qpos. This will cause the planner to timeout.

```
        # this qpos causes several joints to dip below the floor
        env_collision_qpos = [0, 1.5, 0, -1.5, 0, 0, 0]
        self.print_collisions(self.planner.check_for_env_collision(env_collision_qpos))

        print("\n----- env-collision causing planner to timeout -----")
        status, path = self.planner.planner.plan(
            start_state=self_collision_free_qpos, goal_states=[env_collision_qpos]
        )
        print(status, path)
```

Finally, we remove the floor and check for environment collision again. This should return no collision.

```
        self.planner.remove_normal_object("floor")
        self.print_collisions(self.planner.check_for_env_collision(env_collision_qpos))
```

3.1.6 Planning With Fixed Joints

The planner also has the ability to temporarily fix certain joints during planning. The above shows the robot arm move by itself to pick up a red cube before staying in place and letting the base carry the fixed arm. For this tutorial, we will need a different URDF than the one we have used in the previous tutorials. In particular, this URDF has a set of x and y linear tracks that allows the arm to move horizontally. To load this URDF, we do the following:

```
        self.load_robot(urdf_path="./data/panda/panda_on_rail.urdf")
        link_names = [
            "track_x",
```

(continues on next page)

(continued from previous page)

```
            "track_y",
            "panda_link0",
            "panda_link1",
            "panda_link2",
            "panda_link3",
            "panda_link4",
            "panda_link5",
            "panda_link6",
            "panda_link7",
            "panda_hand",
            "panda_leftfinger",
            "panda_rightfinger",
        ]
        joint_names = [
            "move_x",
            "move_y",
            "panda_joint1",
            "panda_joint2",
            "panda_joint3",
            "panda_joint4",
            "panda_joint5",
            "panda_joint6",
            "panda_joint7",
            "panda_finger_joint1",
            "panda_finger_joint2",
        ]
        self.setup_planner(
            urdf_path="./data/panda/panda_on_rail.urdf",
            srdf_path="./data/panda/panda_on_rail.srdf",
            link_names=link_names,
            joint_names=joint_names,
            joint_vel_limits=np.ones(9),
            joint_acc_limits=np.ones(9),
        )
```

The optional *link_names* and *joint_names* parameters used to order the joints and links in a certain way are in this case used to show what the joints of the models are. Then, we set up the planning scene as usual and first move the arm on the track before moving the arm itself to grab the object.

```
        pickup_pose = [0.7, 0, 0.12, 0, 1, 0, 0]
        delivery_pose = [0.4, 0.3, 0.13, 0, 1, 0, 0]

        self.add_point_cloud()
        # also add the target as a collision object so we don't hit it
        fcl_red_cube = fcl.Box([0.04, 0.04, 0.14])
        collision_object = fcl.CollisionObject(fcl_red_cube, Pose(p=[0.7, 0, 0.07]))
        self.planner.planning_world.add_object("target", collision_object)

        # go above the target
        pickup_pose[2] += 0.2
        self.move_in_two_stage(pickup_pose)
```

Notice we have abstracted away how to decouple this motion into two stages. Here is the function definition:

```
    def move_in_two_stage(self, pose, has_attach=False):
        """
        first, we do a full IK but only generate motions for the base
        then, do another partial IK for the arm and generate motions for the arm
        """
        # do a full ik to the pose
        status, goal_qposes = self.planner.IK(pose, self.robot.get_qpos())
        if status != "Success":
            print("IK failed")
            sys.exit(1)
        # now fix arm joints and plan a path to the goal
        result = self.planner.plan_qpos(
            goal_qposes,
            self.robot.get_qpos(),
            time_step=1 / 250,
            fixed_joint_indices=range(2, 9),
        )
        # execute the planned path
        self.follow_path(result)
        result = self.plan_without_base(pose, has_attach)
        # execute the planned path
        self.follow_path(result)
```

The highlighted line is how we ignore the arm joints during planning. We ignore joints 2-9, keeping only joint 0 and 1 active. We then do the same thing except the joints fixed are 0 and 1, and the active joints are 2-9.

3.1.7 Constrained Planning

This is a simple implementation of a constrained planner. It is based on OMPL's projection-based planner. Roughly, OMPL does sampling based planning and projects a joint configuration into a valid configuration using the constrained function we provide. The above gif shows the robot execute two trajectories. The first one generated with constraint that the z-axis of the endeffector pointing downwards. The second one generated without any constraints. We can see that the second trajectory tilts the endeffector sideways.

Defining the Constrained Function

The constrained function is a $R^d \rightarrow R$ function that evaulates to zero when the configuration is valid and non-zero otherwise. In this example, we define a constrained function that evaluates to zero when the z-axis of the endeffector is pointing downwards.

```
        def f(x, out):
            self.planner.robot.set_qpos(x)
            out[0] = (
                self.get_eef_z().dot(np.array([0, 0, -1])) - 0.966
            )  # maintain 15 degrees w.r.t. -z axis
```

Moreover, due to the projection-based method, we also need to provide the jacobian of the constrained function. In this example, we define the jacobian of the constrained function as follows.

```
        def j(x, out):
            full_qpos = self.planner.pad_move_group_qpos(x)
            jac = self.planner.robot.get_pinocchio_model().compute_single_link_jacobian(
                full_qpos, len(self.planner.move_group_joint_indices) - 1
            )
            rot_jac = jac[3:, self.planner.move_group_joint_indices]
            for i in range(len(self.planner.move_group_joint_indices)):
                out[i] = np.cross(rot_jac[:, i], self.get_eef_z()).dot(
                    np.array([0, 0, -1])
                )
```

One can usually calculate the jacobian of the constraint by manipulating the jacobian of the forward kinematics. We need the jacobian calculation to be fast or else the planner will be slow. In the case above, we used the single link jacobian of the endeffector and used its rotational part to calculate how much the z-axis of the endeffector is changing.

Using the Constrained Planner

The interface to the constrained planner is just some parameters when calling the planning function. We need to pass in the constrained function as well as its jacobian. Optinally, pass in the tolerance for the projection.

```
        print(
            "with constraint. all movements roughly maintain 15 degrees w.r.t. -z axis"
        )
        for pose in poses:
            result = self.planner.plan_pose(
                pose,
                self.robot.get_qpos(),
                time_step=1 / 250,
                constraint_function=self.make_f(),
                constraint_jacobian=self.make_j(),
                constraint_tolerance=0.05,
            )
            if result["status"] != "Success":
                print(result["status"])
                return -1
            self.follow_path(result)
```

CHAPTER

FOUR

API REFERENCE

4.1 API Reference

4.1.1 Planner

class mplib.**Planner**(
urdf: str | Path,
move_group: str,
***,
srdf: str | Path | None = None,
new_package_keyword: str = '',
use_convex: bool = False,
user_link_names: Sequence[str] = [],
user_joint_names: Sequence[str] = [],
joint_vel_limits: Sequence[float] | ndarray | None = None,
joint_acc_limits: Sequence[float] | ndarray | None = None,
objects: list[FCLObject] = [],
verbose: bool = False,
)

Bases: `object`

Motion planner

__init__(
urdf: str | Path,
move_group: str,
***,
srdf: str | Path | None = None,
new_package_keyword: str = '',
use_convex: bool = False,
user_link_names: Sequence[str] = [],
user_joint_names: Sequence[str] = [],
joint_vel_limits: Sequence[float] | ndarray | None = None,
joint_acc_limits: Sequence[float] | ndarray | None = None,
objects: list[FCLObject] = [],
verbose: bool = False,
)

Motion planner for robots.

References

http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/urdf_srdf/urdf_srdf_tutorial.html

Parameters

- **urdf** – Unified Robot Description Format file.
- **move_group** – target link to move, usually the end-effector.
- **srdf** – Semantic Robot Description Format file.
- **new_package_keyword** – the string to replace `package://` keyword
- **use_convex** – if True, load collision mesh as convex mesh. If mesh is not convex, a `RuntimeError` will be raised.
- **user_link_names** – names of links, the order matters. If empty, all links will be used.
- **user_joint_names** – names of the joints to plan. If empty, all active joints will be used.
- **joint_vel_limits** – maximum joint velocities for time parameterization, which should have the same length as `self.move_group_joint_indices`
- **joint_acc_limits** – maximum joint accelerations for time parameterization, which should have the same length as `self.move_group_joint_indices`
- **objects** – list of FCLObject as non-articulated collision objects
- **verbose** – if True, print verbose logs for debugging

wrap_joint_limit(*qpos: ndarray*) → bool

Checks if the joint configuration can be wrapped to be within the joint limits. For revolute joints, the joint angle is wrapped to be within [q_min, q_min+2*pi)

Parameters
qpos – joint positions, angles of revolute joints might be modified. If not within_limits (returns False), qpos might not be fully wrapped.

Returns
whether qpos can be wrapped to be within the joint limits.

pad_move_group_qpos(*qpos*, *articulation=None*)

If qpos contains only the move_group joints, return qpos padded with current values of the remaining joints of articulation. Otherwise, verify number of joints and return.

Parameters

- **qpos** – joint positions
- **articulation** – the articulation to get qpos from. If None, use self.robot

Returns
joint positions with full dof

check_for_collision(
collision_function,
state: ndarray | None = None,
) → list[*WorldCollisionResult*]

Helper function to check for collision

Parameters
state – all planned articulations qpos state. If None, use current qpos.

Returns
A list of collisions.

check_for_self_collision(
state: ndarray | None = None,
) → list[*WorldCollisionResult*]

Check if the robot is in self-collision.

Parameters
state – all planned articulations qpos state. If None, use current qpos.

Returns
A list of collisions.

check_for_env_collision(
state: ndarray | None = None,
) → list[*WorldCollisionResult*]

Check if the robot is in collision with the environment

Parameters
state – all planned articulations qpos state. If None, use current qpos.

Returns
A list of collisions.

IK(
goal_pose: Pose,
start_qpos: ndarray,
mask: Sequence[bool] | ndarray | None = None,
***,
n_init_qpos: int = 20,
threshold: float = 1e-3,
return_closest: bool = False,
verbose: bool = False,
) → tuple[str, list[ndarray] | ndarray | None]

Compute inverse kinematics

Parameters

- **goal_pose** – goal pose
- **start_qpos** – initial configuration, (ndof,) np.floating np.ndarray.
- **mask** – qpos mask to disable IK sampling, (ndof,) bool np.ndarray.
- **n_init_qpos** – number of random initial configurations to sample.
- **threshold** – distance threshold for marking sampled IK as success. distance is position error norm + quaternion error norm.
- **return_closest** – whether to return the qpos that is closest to start_qpos, considering equivalent joint values.
- **verbose** – whether to print collision info if any collision exists.

Returns

(status, q_goals)

status: IK status, “Success” if succeeded.

q_goals: list of sampled IK qpos, (ndof,) np.floating np.ndarray.
IK is successful if q_goals is not None. If return_closest, q_goals is np.ndarray if successful and None if not successful.

TOPP(*path*, *step=0.1*, *verbose=False*)

Time-Optimal Path Parameterization

Parameters

- **path** – numpy array of shape (n, dof)
- **step** – step size for the discretization
- **verbose** – if True, will print the log of TOPPRA

update_point_cloud(*points*, *resolution=1e-3*, *name='scene_pcd'*)

Adds a point cloud as a collision object with given name to world. If the `name` is the same, the point cloud is simply updated.

Parameters

- **points** – points, numpy array of shape (n, 3)
- **resolution** – resolution of the point OcTree
- **name** – name of the point cloud collision object

remove_point_cloud(*name='scene_pcd'*) → bool

Removes the point cloud collision object with given name

Parameters
name – name of the point cloud collision object

Returns
`True` if success, `False` if the non-articulation object with given name does not exist

update_attached_object(
collision_geometry: CollisionGeometry,
pose: Pose,
name='attached_geom',
art_name=None,
link_id=-1,
)

Attach given object (w/ collision geometry) to specified link of articulation

Parameters

- **collision_geometry** – FCL collision geometry
- **pose** – attaching pose (relative pose from attached link to object)
- **name** – name of the attached geometry.
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_attached_sphere(
radius: float,
pose: Pose,
art_name=None,
link_id=-1,
)

Attach a sphere to some link

Parameters

- **radius** – radius of the sphere
- **pose** – attaching pose (relative pose from attached link to object)
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_attached_box(
size: Sequence[float] | ndarray[tuple[Literal[3], Literal[1]], dtype[floating]],
pose: Pose,
art_name=None,
link_id=-1,
)

Attach a box to some link

Parameters

- **size** – box side length
- **pose** – attaching pose (relative pose from attached link to object)
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_attached_mesh(
mesh_path: str,
pose: Pose,
art_name=None,
link_id=-1,
)

Attach a mesh to some link

Parameters

- **mesh_path** – path to a mesh file
- **pose** – attaching pose (relative pose from attached link to object)
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

detach_object(*name='attached_geom'*, *also_remove=False*) → bool

Detact the attached object with given name

Parameters

- **name** – object name to detach
- **also_remove** – whether to also remove object from world

Returns

`True` if success, `False` if the object with given name is not attached

set_base_pose(*pose:* Pose)

tell the planner where the base of the robot is w.r.t the world frame

Parameters
pose – pose of the base

remove_object(*name*) → bool

returns true if the object was removed, false if it was not found

plan_qpos(
goal_qposes: list[ndarray],
current_qpos: ndarray,
*,
time_step: float = 0.1,
rrt_range: float = 0.1,
planning_time: float = 1,
fix_joint_limits: bool = True,
fixed_joint_indices: list[int] | None = None,
simplify: bool = True,
constraint_function: Callable[[ndarray, ndarray], None] | None = None,
constraint_jacobian: Callable[[ndarray, ndarray], None] | None = None,
constraint_tolerance: float = 1e-3,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

Plan a path from a specified joint position to a goal pose

Parameters

- **goal_qposes** – list of target joint configurations, [(ndof,)]
- **current_qpos** – current joint configuration (either full or move_group joints)
- **mask** – mask for IK. When set, the IK will leave certain joints out of planning
- **time_step** – time step for TOPP
- **rrt_range** – step size for RRT
- **planning_time** – time limit for RRT
- **fix_joint_limits** – if True, will clip the joint configuration to be within the joint limits
- **simplify** – whether the planned path will be simplified. (constained planning does not support simplification)
- **constraint_function** – evals to 0 when constraint is satisfied
- **constraint_jacobian** – jacobian of constraint_function
- **constraint_tolerance** – tolerance for constraint_function
- **fixed_joint_indices** – list of indices of joints that are fixed during planning
- **verbose** – if True, will print the log of OMPL and TOPPRA

plan_pose(
goal_pose: Pose,
current_qpos: ndarray,
mask: list[bool] | ndarray | None = None,
*,
time_step: float = 0.1,
rrt_range: float = 0.1,
planning_time: float = 1,
fix_joint_limits: bool = True,
wrt_world: bool = True,

simplify: bool = True,
constraint_function: Callable | None = None,
constraint_jacobian: Callable | None = None,
constraint_tolerance: float = 1e-3,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

plan from a start configuration to a goal pose of the end-effector

Parameters

- **goal_pose** – pose of the goal
- **current_qpos** – current joint configuration (either full or move_group joints)
- **mask** – if the value at a given index is True, the joint is *not* used in the IK
- **time_step** – time step for TOPPRA (time parameterization of path)
- **rrt_range** – step size for RRT
- **planning_time** – time limit for RRT
- **fix_joint_limits** – if True, will clip the joint configuration to be within the joint limits
- **wrt_world** – if true, interpret the target pose with respect to the world frame instead of the base frame
- **verbose** – if True, will print the log of OMPL and TOPPRA

plan_screw(
goal_pose: Pose,
current_qpos: ndarray,
*,
qpos_step: float = 0.1,
time_step: float = 0.1,
wrt_world: bool = True,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

Plan from a start configuration to a goal pose of the end-effector using screw motion

Parameters

- **goal_pose** – pose of the goal
- **current_qpos** – current joint configuration (either full or move_group joints)
- **qpos_step** – size of the random step
- **time_step** – time step for the discretization
- **wrt_world** – if True, interpret the target pose with respect to the world frame instead of the base frame
- **verbose** – if True, will print the log of TOPPRA

4.1.2 PlanningWorld

class mplib.**PlanningWorld**

Bases: pybind11_object

Planning world for collision checking

Mimicking MoveIt2's planning_scene::PlanningScene, collision_detection::World, moveit::core::RobotState, collision_detection::CollisionEnv

https://moveit.picknik.ai/main/api/html/classplanning__scene_1_1PlanningScene.html https://moveit.picknik.ai/main/api/html/classcollision__detection_1_1World.html https://moveit.picknik.ai/main/api/html/classmoveit_1_1core_1_1RobotState.html https://moveit.picknik.ai/main/api/html/classcollision__detection_1_1CollisionEnv.html

__init__(
self: mplib.pymp.PlanningWorld,
*articulations: list[*mplib.pymp.ArticulatedModel*]*,
*objects: list[*mplib.pymp.collision_detection.fcl.FCLObject*] = []*,
) → None

Constructs a PlanningWorld with given (planned) articulations and objects

Parameters

- **articulations** – list of planned articulated models
- **objects** – list of non-articulated collision objects

add_articulation(
self: mplib.pymp.PlanningWorld,
model: mplib.pymp.ArticulatedModel,
planned: bool = False,
) → None

Adds an articulation (ArticulatedModelPtr) to world

Parameters

- **model** – articulated model to be added
- **planned** – whether the articulation is being planned

add_object(**args*, ***kwargs*)

Overloaded function.

1. add_object(self: mplib.pymp.PlanningWorld, fcl_obj: mplib.pymp.collision_detection.fcl.FCLObject) -> None

Adds an non-articulated object containing multiple collision objects (FCLObjectPtr) to world

Parameters
fcl_obj – FCLObject to be added

2. add_object(self: mplib.pymp.PlanningWorld, name: str, collision_object: mplib.pymp.collision_detection.fcl.CollisionObject) -> None

Adds an non-articulated object (CollisionObjectPtr) with given name to world

Parameters

- **name** – name of the collision object
- **collision_object** – collision object to be added

add_point_cloud(
self: mplib.pymp.PlanningWorld,
name: str,
vertices: numpy.ndarray[numpy.float64[m, 3]],
resolution: float = 0.01,
) → None

Adds a point cloud as a collision object with given name to world

Parameters

- **name** – name of the point cloud collision object
- **vertices** – point cloud vertices matrix
- **resolution** – resolution of the point in `octomap::OcTree`

attach_box(
self: mplib.pymp.PlanningWorld,
size: numpy.ndarray[numpy.float64[3, 1]],
art_name: str,
link_id: int,
pose: mplib.pymp.Pose,
) → None

Attaches given box to specified link of articulation (auto touch_links)

Parameters

- **size** – box side length
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

attach_mesh(
self: mplib.pymp.PlanningWorld,
mesh_path: str,
art_name: str,
link_id: int,
pose: mplib.pymp.Pose,
*,
convex: bool = False,
) → None

Attaches given mesh to specified link of articulation (auto touch_links)

Parameters

- **mesh_path** – path to a mesh file
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)
- **convex** – whether to load mesh as a convex mesh. Default: `False`.

attach_object(**args*, ***kwargs*)

Overloaded function.

1. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int, touch_links: list[str]) -> None

Attaches existing non-articulated object to specified link of articulation at its current pose. If the object is currently attached, disallow collision between the object and previous touch_links.

Updates **acm_** to allow collisions between attached object and touch_links.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **touch_links** – link names that the attached object touches

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

2. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int) -> None

Attaches existing non-articulated object to specified link of articulation at its current pose. If the object is not currently attached, automatically sets touch_links as the name of self links that collide with the object in the current state.

Updates **acm_** to allow collisions between attached object and touch_links.

If the object is already attached, the touch_links of the attached object is preserved and **acm_** remains unchanged.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

3. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int, pose: mplib.pymp.Pose, touch_links: list[str]) -> None

Attaches existing non-articulated object to specified link of articulation at given pose. If the object is currently attached, disallow collision between the object and previous touch_links.

Updates **acm_** to allow collisions between attached object and touch_links.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)
- **touch_links** – link names that the attached object touches

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

4. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int, pose: mplib.pymp.Pose) -> None

Attaches existing non-articulated object to specified link of articulation at given pose. If the object is not currently attached, automatically sets touch_links as the name of self links that collide with the object in the current state.

Updates **acm_** to allow collisions between attached object and touch_links.

If the object is already attached, the touch_links of the attached object is preserved and **acm_** remains unchanged.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

5. attach_object(self: mplib.pymp.PlanningWorld, name: str, p_geom: mplib.pymp.collision_detection.fcl.CollisionGeometry, art_name: str, link_id: int, pose: mplib.pymp.Pose, touch_links: list[str]) -> None

Attaches given object (w/ p_geom) to specified link of articulation at given pose. This is done by removing the object and then adding and attaching object. As a result, all previous **acm_** entries with the object are removed

Parameters

- **name** – name of the non-articulated object to attach
- **p_geom** – pointer to a CollisionGeometry object
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)
- **touch_links** – link names that the attached object touches

6. attach_object(self: mplib.pymp.PlanningWorld, name: str, p_geom: mplib.pymp.collision_detection.fcl.CollisionGeometry, art_name: str, link_id: int, pose: mplib.pymp.Pose) -> None

Attaches given object (w/ p_geom) to specified link of articulation at given pose. This is done by removing the object and then adding and attaching object. As a result, all previous **acm_** entries with the object are removed. Automatically sets touch_links as the name of self links that collide with the object in the current state (auto touch_links).

Parameters

- **name** – name of the non-articulated object to attach
- **p_geom** – pointer to a CollisionGeometry object

- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

attach_sphere(
self: mplib.pymp.PlanningWorld,
radius: float,
art_name: str,
link_id: int,
pose: mplib.pymp.Pose,
) → None

Attaches given sphere to specified link of articulation (auto touch_links)

Parameters

- **radius** – sphere radius
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

check_collision(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f18bf0>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check full collision (calls `checkSelfCollision()` and `checkRobotCollision()`)

Parameters
request – collision request params.

Returns
List of `WorldCollisionResult` objects

check_robot_collision(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fd0bd0f70>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check collision with other scene bodies in the world (planned articulations with attached objects collide against unplanned articulations and scene objects)

Parameters
request – collision request params.

Returns
List of `WorldCollisionResult` objects

check_self_collision(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc9b18af0>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check for self collision (including planned articulation self-collision, planned articulation-attach collision, attach-attach collision)

Parameters
request – collision request params.

Returns
List of `WorldCollisionResult` objects

`detach_object(`
self: mplib.pymp.PlanningWorld,
name: str,
also_remove: bool = False,
`) → bool`

Detaches object with given name. Updates **acm_** to disallow collision between the object and touch_links.

Parameters

- **name** – name of the non-articulated object to detach
- **also_remove** – whether to also remove object from world

Returns
`True` if success, `False` if the object with given name is not attached

`distance(`
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fc8f125b0>,
`) →` *mplib.pymp.collision_detection.WorldDistanceResult*

Compute the minimum distance-to-all-collision (calls `distanceSelf()` and `distanceRobot()`)

Parameters
request – distance request params.

Returns
a `WorldDistanceResult` object

`distance_robot(`
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fd0b66270>,
`) →` *mplib.pymp.collision_detection.WorldDistanceResult*

Compute the minimum distance-to-collision between a robot and the world

Parameters
request – distance request params.

Returns
a `WorldDistanceResult` object

`distance_self(`
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fd0b978b0>,
`) →` *mplib.pymp.collision_detection.WorldDistanceResult*

Get the minimum distance to self-collision given the robot in current state

Parameters
request – distance request params.

Returns
a `WorldDistanceResult` object

distance_to_collision(*self:* mplib.pymp.PlanningWorld) → float

Compute the minimum distance-to-all-collision. Calls `distance()`.

Note that this is different from MoveIt2's `planning_scene::PlanningScene::distanceToCollision()` where self-collisions are ignored.

Returns
minimum distance-to-all-collision

distance_to_robot_collision(*self:* mplib.pymp.PlanningWorld) → float

The distance between the robot model at current state to the nearest collision (ignoring self-collisions). Calls `distanceRobot()`.

Returns
minimum distance-to-robot-collision

distance_to_self_collision(*self:* mplib.pymp.PlanningWorld) → float

The minimum distance to self-collision given the robot in current state. Calls `distanceSelf()`.

Returns
minimum distance-to-self-collision

get_allowed_collision_matrix(
self: mplib.pymp.PlanningWorld,
) → *mplib.pymp.collision_detection.AllowedCollisionMatrix*

Get the current allowed collision matrix

get_articulation(
self: mplib.pymp.PlanningWorld,
name: str,
) → *mplib.pymp.ArticulatedModel*

Gets the articulation (ArticulatedModelPtr) with given name

Parameters
name – name of the articulated model

Returns
the articulated model with given name or `None` if not found.

get_articulation_names(*self:* mplib.pymp.PlanningWorld) → list[str]

Gets names of all articulations in world (unordered)

get_attached_object(
self: mplib.pymp.PlanningWorld,
name: str,
) → *mplib.pymp.AttachedBody*

Gets the attached body (AttachedBodyPtr) with given name

Parameters
name – name of the attached body

Returns
the attached body with given name or `None` if not found.

get_object(
self: mplib.pymp.PlanningWorld,
name: str,
) → *mplib.pymp.collision_detection.fcl.FCLObject*

Gets the non-articulated object (`FCLObjectPtr`) with given name

Parameters
name – name of the non-articulated object

Returns
the object with given name or `None` if not found.

`get_object_names`(*self:* mplib.pymp.PlanningWorld) → list[str]

Gets names of all objects in world (unordered)

`get_planned_articulations`(
self: mplib.pymp.PlanningWorld,
) → list[*mplib.pymp.ArticulatedModel*]

Gets all planned articulations (ArticulatedModelPtr)

`has_articulation`(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Check whether the articulation with given name exists

Parameters
name – name of the articulated model

Returns
`True` if exists, `False` otherwise.

`has_object`(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Check whether the non-articulated object with given name exists

Parameters
name – name of the non-articulated object

Returns
`True` if exists, `False` otherwise.

`is_articulation_planned`(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Check whether the articulation with given name is being planned

Parameters
name – name of the articulated model

Returns
`True` if exists, `False` otherwise.

`is_object_attached`(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Check whether the non-articulated object with given name is attached

Parameters
name – name of the non-articulated object

Returns
`True` if it is attached, `False` otherwise.

`is_state_colliding`(*self:* mplib.pymp.PlanningWorld) → bool

Check if the current state is in collision (with the environment or self collision).

Returns
`True` if collision exists

`print_attached_body_pose`(*self:* mplib.pymp.PlanningWorld) → None

Prints global pose of all attached bodies

remove_articulation(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Removes the articulation with given name if exists. Updates **acm_**

Parameters
name – name of the articulated model

Returns
`True` if success, `False` if articulation with given name does not exist

remove_object(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Removes (and detaches) the collision object with given name if exists. Updates **acm_**

Parameters
name – name of the non-articulated collision object

Returns
`True` if success, `False` if the non-articulated object with given name does not exist

set_articulation_planned(
self: mplib.pymp.PlanningWorld,
name: str,
planned: bool,
) → None

Sets articulation with given name as being planned

Parameters

- **name** – name of the articulated model
- **planned** – whether the articulation is being planned

Raises
ValueError – if the articulation with given name does not exist

set_qpos(
self: mplib.pymp.PlanningWorld,
name: str,
qpos: numpy.ndarray[numpy.float64[m, 1]],
) → None

Set qpos of articulation with given name

Parameters

- **name** – name of the articulated model
- **qpos** – joint angles of the *movegroup only* // FIXME: double check

set_qpos_all(
self: mplib.pymp.PlanningWorld,
state: numpy.ndarray[numpy.float64[m, 1]],
) → None

Set qpos of all planned articulations

4.1.3 ArticulatedModel

class mplib.**ArticulatedModel**

Bases: pybind11_object

Supports initialization from URDF and SRDF files, and provides access to underlying Pinocchio and FCL models.

__init__(
self: mplib.pymp.ArticulatedModel,
urdf_filename: str,
srdf_filename: str,
***,
name: str = None,
gravity: numpy.ndarray[numpy.float64[3, 1]] = array([0., 0., -9.81]),
link_names: list[str] = [],
joint_names: list[str] = [],
convex: bool = False,
verbose: bool = False,
) → None

Construct an articulated model from URDF and SRDF files.

Parameters

- **urdf_filename** – path to URDF file, can be relative to the current working directory
- **srdf_filename** – path to SRDF file, we use it to disable self-collisions
- **name** – name of the articulated model to override URDF robot name attribute
- **gravity** – gravity vector, by default is `[0, 0, -9.81]` in -z axis
- **link_names** – list of links that are considered for planning
- **joint_names** – list of joints that are considered for planning
- **convex** – use convex decomposition for collision objects. Default: `False`.
- **verbose** – print debug information. Default: `False`.

property base_pose

The base (root) pose of the robot

static create_from_urdf_string(
urdf_string: str,
srdf_string: str,
*collision_links: list[*mplib.pymp.collision_detection.fcl.FCLObject*]*,
***,
name: str = None,
gravity: numpy.ndarray[numpy.float64[3, 1]] = array([0., 0., -9.81]),
link_names: list[str] = [],
joint_names: list[str] = [],
verbose: bool = False,
) → *mplib.pymp.ArticulatedModel*

Constructs an ArticulatedModel from URDF/SRDF strings and collision links

Parameters

- **urdf_string** – URDF string (without visual/collision elements for links)
- **srdf_string** – SRDF string (only disable_collisions element)

- **collision_links** – Vector of collision links as FCLObjectPtr. Format is: `[FCLObjectPtr, ...]`. The collision objects are at the shape's local_pose.
- **name** – name of the articulated model to override URDF robot name attribute
- **gravity** – gravity vector, by default is `[0, 0, -9.81]` in -z axis
- **link_names** – list of links that are considered for planning
- **joint_names** – list of joints that are considered for planning
- **verbose** – print debug information. Default: `False`.

Returns
a unique_ptr to ArticulatedModel

get_base_pose(*self:* mplib.pymp.ArticulatedModel) → *mplib.pymp.Pose*

Get the base (root) pose of the robot.

Returns
base pose of the robot

get_fcl_model(
self: mplib.pymp.ArticulatedModel,
) → *mplib.pymp.collision_detection.fcl.FCLModel*

Get the underlying FCL model.

Returns
FCL model used for collision checking

get_move_group_end_effectors(
self: mplib.pymp.ArticulatedModel,
) → list[str]

Get the end effectors of the move group.

Returns
list of end effectors of the move group

get_move_group_joint_indices(
self: mplib.pymp.ArticulatedModel,
) → list[int]

Get the joint indices of the move group.

Returns
list of user joint indices of the move group

get_move_group_joint_names(
self: mplib.pymp.ArticulatedModel,
) → list[str]

Get the joint names of the move group.

Returns
list of joint names of the move group

get_move_group_qpos_dim(*self:* mplib.pymp.ArticulatedModel) → int

Get the dimension of the move group qpos.

Returns
dimension of the move group qpos

get_name(*self:* mplib.pymp.ArticulatedModel) → str

Get name of the articulated model.

Returns
name of the articulated model

get_pinocchio_model(
self: mplib.pymp.ArticulatedModel,
) → *mplib.pymp.kinematics.pinocchio.PinocchioModel*

Get the underlying Pinocchio model.

Returns
Pinocchio model used for kinematics and dynamics computations

get_pose(*self:* mplib.pymp.ArticulatedModel) → *mplib.pymp.Pose*

Get the base (root) pose of the robot.

Returns
base pose of the robot

get_qpos(
self: mplib.pymp.ArticulatedModel,
) → numpy.ndarray[numpy.float64[m, 1]]

Get the current joint position of all active joints inside the URDF.

Returns
current qpos of all active joints

get_user_joint_names(*self:* mplib.pymp.ArticulatedModel) → list[str]

Get the joint names that the user has provided for planning.

Returns
list of joint names of the user

get_user_link_names(*self:* mplib.pymp.ArticulatedModel) → list[str]

Get the link names that the user has provided for planning.

Returns
list of link names of the user

property name

Name of the articulated model

property pose

The base (root) pose of the robot

property qpos

Current qpos of all active joints

set_base_pose(
self: mplib.pymp.ArticulatedModel,
pose: mplib.pymp.Pose,
) → None

Set the base pose of the robot and update all collision links in the `FCLModel`.

Parameters
pose – base pose of the robot

set_move_group(**args*, ***kwargs*)

Overloaded function.

1. set_move_group(self: mplib.pymp.ArticulatedModel, end_effector: str) -> None

Set the move group, i.e. the chain ending in end effector for which to compute the forward kinematics for all subsequent queries.

Parameters
end_effector – name of the end effector link

2. set_move_group(self: mplib.pymp.ArticulatedModel, end_effectors: list[str]) -> None

Set the move group but we have multiple end effectors in a chain. I.e., Base –> EE1 –> EE2 –> … –> EEn

Parameters
end_effectors – list of links extending to the end effector

set_pose(
self: mplib.pymp.ArticulatedModel,
pose: mplib.pymp.Pose,
) → None

Set the base pose of the robot and update all collision links in the `FCLModel`.

Parameters
pose – base pose of the robot

set_qpos(
self: mplib.pymp.ArticulatedModel,
qpos: numpy.ndarray[numpy.float64[m, 1]],
full: bool = False,
) → None

Set the current joint positions and update all collision links in the `FCLModel`.

Parameters

- **qpos** – current qpos of all active joints or just the move group joints
- **full** – whether to set the full qpos or just the move group qpos. If full is `False`, we will pad the missing joints with current known qpos. The default is `False`

update_SRDF(*self:* mplib.pymp.ArticulatedModel, *SRDF: str*) → None

Update the SRDF file to disable self-collisions.

Parameters
srdf – path to SRDF file, can be relative to the current working directory

4.1.4 AttachedBody

class mplib.**AttachedBody**

Bases: `pybind11_object`

Object defining bodies that can be attached to robot links. This is useful when handling objects picked up by the robot.

Mimicking MoveIt2's `moveit::core::AttachedBody`

https://moveit.picknik.ai/main/api/html/classmoveit_1_1core_1_1AttachedBody.html

__init__(
self: mplib.pymp.AttachedBody,
name: str,
object: mplib.pymp.collision_detection.fcl.FCLObject,
attached_articulation: mplib.pymp.ArticulatedModel,
attached_link_id: int,
pose: mplib.pymp.Pose,
touch_links: list[str] = [],
) → None

Construct an attached body for a specified link.

Parameters

- **name** – name of the attached body
- **object** – collision object of the attached body
- **attached_articulation** – robot articulated model to attach to
- **attached_link_id** – id of the articulation link to attach to
- **pose** – attached pose (relative pose from attached link to object)
- **touch_links** – the link names that the attached body touches

get_attached_articulation(
self: mplib.pymp.AttachedBody,
) → *mplib.pymp.ArticulatedModel*

Gets the articulation that this body is attached to

get_attached_link_global_pose(
self: mplib.pymp.AttachedBody,
) → *mplib.pymp.Pose*

Gets the global pose of the articulation link that this body is attached to

get_attached_link_id(*self:* mplib.pymp.AttachedBody) → int

Gets the articulation link id that this body is attached to

get_global_pose(*self:* mplib.pymp.AttachedBody) → *mplib.pymp.Pose*

Gets the global pose of the attached object

get_name(*self:* mplib.pymp.AttachedBody) → str

Gets the attached object name

get_object(
self: mplib.pymp.AttachedBody,
) → *mplib.pymp.collision_detection.fcl.FCLObject*

Gets the attached object (FCLObjectPtr)

get_pose(*self:* mplib.pymp.AttachedBody) → *mplib.pymp.Pose*

Gets the attached pose (relative pose from attached link to object)

get_touch_links(*self:* mplib.pymp.AttachedBody) → list[str]

Gets the link names that the attached body touches

property pose

The attached pose (relative pose from attached link to object)

set_pose(*self:* mplib.pymp.AttachedBody, *pose:* mplib.pymp.Pose) → None

Sets the attached pose (relative pose from attached link to object)

set_touch_links(
self: mplib.pymp.AttachedBody,
touch_links: list[str],
) → None

Sets the link names that the attached body touches

update_pose(*self:* mplib.pymp.AttachedBody) → None

Updates the global pose of the attached object using current state

4.1.5 Pose

class mplib.**Pose**

Bases: pybind11_object

Pose stored as a unit quaternion and a position vector

This struct is intended to be used only for interfacing with Python. Internally, Pose is converted to and stored as Eigen::Isometry3 which is used by all computations.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.Pose) -> None

Constructs a default Pose with p = (0,0,0) and q = (1,0,0,0)

2. __init__(self: mplib.pymp.Pose, p: numpy.ndarray[numpy.float64[3, 1]] = array([0., 0., 0.]), q: numpy.ndarray[numpy.float64[4, 1]] = array([1., 0., 0., 0.])) -> None

Constructs a Pose with given position and quaternion

Parameters

- **p** – position, format: (x, y, z)
- **q** – quaternion (can be unnormalized), format: (w, x, y, z)

3. __init__(self: mplib.pymp.Pose, matrix: numpy.ndarray[numpy.float64[4, 4]]) -> None

Constructs a Pose with given transformation matrix (4x4 np.ndarray with np.float64 dtype)

Parameters
matrix – a 4x4 np.float64 np.ndarray transformation matrix

4. __init__(self: mplib.pymp.Pose, obj: object) -> None

Constructs a Pose with given Python object that has p and q attributes (e.g., sapien.Pose) or a 4x4 np.ndarray transformation matrix.

Parameters
obj – a Pose-like object with p and q attributes or a 4x4 np.ndarray transformation matrix

distance(*self:* mplib.pymp.Pose, *other:* mplib.pymp.Pose) → float

Computes the distance between two poses by ``norm(p1.p - p2.p) + min(norm(p1.q - p2.q), norm(p1.q + p2.q))`.

The quaternion part has range [0, sqrt(2)].

Parameters
other – the other pose

Returns
the distance between the two poses

get_p(*self:* mplib.pymp.Pose) → numpy.ndarray[numpy.float64[3, 1]]

Gets the position part of the Pose

Returns
position, format: (x, y, z)

get_q(*self:* mplib.pymp.Pose) → numpy.ndarray[numpy.float64[4, 1]]

Gets the quaternion part of the Pose

Returns
quaternion, format: (w, x, y, z)

inv(*self:* mplib.pymp.Pose) → *mplib.pymp.Pose*

Get the inserse Pose

Returns
the inverse Pose

property p

Position part of the Pose (x, y, z)

property q

Quaternion part of the Pose (w, x, y, z)

set_p(*self:* mplib.pymp.Pose, *p: numpy.ndarray[numpy.float64[3, 1]]*) → None

Sets the position part of the Pose

Parameters
p – position, format: (x, y, z)

set_q(*self:* mplib.pymp.Pose, *q: numpy.ndarray[numpy.float64[4, 1]]*) → None

Sets the quaternion part of the Pose

Parameters
q – quaternion (can be unnormalized), format: (w, x, y, z)

to_transformation_matrix(
self: mplib.pymp.Pose,
) → numpy.ndarray[numpy.float64[4, 4]]

Constructs a transformation matrix from this Pose

Returns
a 4x4 transformation matrix

4.1.6 set_global_seed

mplib.**set_global_seed**(*seed: int*) → None

Sets the global seed for MPlib (`std::srand()`, OMPL's RNG, and FCL's RNG).

Parameters

seed – the random seed value

4.1.7 urdf_utils

mplib.urdf_utils.**compute_default_collisions**(
robot: ArticulatedModel,
*,
num_samples=100000,
verbose=False,
) → str

Compute default collision pairs for generating SRDF.

This function mimics MoveIt2's `moveit_setup::srdf_setup::computeDefaultCollisions()`

Reference: https://moveit.picknik.ai/main/api/html/namespacemoveit__setup_1_1srdf__setup.html#a2812f73b447d838cd7dba1b0ee1a0c95

Parameters

- **robot** – an `ArticulatedModel`
- **num_samples** – number of samples to find the link that will always collide
- **verbose** – print debug info

Returns

SRDF content as an XML string

mplib.urdf_utils.**replace_urdf_package_keyword**(
urdf_path: str | Path,
new_package_keyword: str = '',
) → Path

Some ROS URDF files use package:// keyword to refer the package dir. Replace it with the given string (default is empty)

Parameters

- **urdf_path** – Path to a Unified Robot Description Format file.
- **new_package_keyword** – the string to replace `package://` keyword

Returns

Path to the modified URDF file

mplib.urdf_utils.**generate_srdf**(
urdf_path: str | Path,
new_package_keyword: str = '',
*,
num_samples=100000,
verbose=False,
) → Path

Generate SRDF from URDF similar to MoveIt2's setup assistant.

Parameters

- **urdf_path** – Path to a Unified Robot Description Format file.
- **new_package_keyword** – the string to replace `package://` keyword
- **num_samples** – number of samples to find the link that will always collide
- **verbose** – print debug info

Returns

Path to the generated SRDF file

4.1.8 sapien_utils

```
class mplib.sapien_utils.SapienPlanningWorld(
    sim_scene: Scene,
    planned_articulations: list[PhysxArticulation] = [],
)
```

Bases: *PlanningWorld*

```
__init__(
    sim_scene: Scene,
    planned_articulations: list[PhysxArticulation] = [],
)
```

Creates an mplib.PlanningWorld from a sapien.Scene.

Parameters

planned_articulations – list of planned articulations.

```
update_from_simulation(
    *,
    update_attached_object: bool = True,
) → None
```

Updates PlanningWorld's articulations/objects pose with current Scene state. Note that shape's local_pose is not updated. If those are changed, please recreate a new SapienPlanningWorld instance.

Parameters

update_attached_object – whether to update the attached pose of all attached objects

```
check_collision_between(
    obj_A: PhysxArticulation | Entity,
    obj_B: PhysxArticulation | Entity,
    *,
    acm: AllowedCollisionMatrix = AllowedCollisionMatrix(),
) → list[WorldCollisionResult]
```

Check collision between two objects, which can either be a PhysxArticulation or an Entity.

Parameters

- **obj_A** – object A to check for collision.
- **obj_B** – object B to check for collision.
- **acm** – allowed collision matrix.

Returns

a list of WorldCollisionResult. Empty if there's no collision.

distance_between(
obj_A: PhysxArticulation | Entity,
obj_B: PhysxArticulation | Entity,
***,
acm: AllowedCollisionMatrix = *AllowedCollisionMatrix()*,
return_distance_only: bool = True,
) → *WorldDistanceResult* | float

Check distance-to-collision between two objects, which can either be a PhysxArticulation or an Entity.

Parameters

- **obj_A** – object A to check for collision.
- **obj_B** – object B to check for collision.
- **acm** – allowed collision matrix.
- **return_distance_only** – if True, return distance only.

Returns
a WorldDistanceResult or a float if return_distance_only==True.

static convert_physx_component(
comp: PhysxRigidBaseComponent,
) → *FCLObject* | None

Converts a SAPIEN physx.PhysxRigidBaseComponent to an FCLObject. All shapes in the returned FCLObject are already set at their world poses.

Parameters
comp – a SAPIEN physx.PhysxRigidBaseComponent.

Returns
an FCLObject containing all collision shapes in the Physx component. If the component has no collision shapes, return `None`.

add_articulation(
self: mplib.pymp.PlanningWorld,
model: mplib.pymp.ArticulatedModel,
planned: bool = False,
) → None

Adds an articulation (ArticulatedModelPtr) to world

Parameters

- **model** – articulated model to be added
- **planned** – whether the articulation is being planned

add_object(**args*, ***kwargs*)

Overloaded function.

1. add_object(self: mplib.pymp.PlanningWorld, fcl_obj: mplib.pymp.collision_detection.fcl.FCLObject) -> None

Adds an non-articulated object containing multiple collision objects (`FCLObjectPtr`) to world

Parameters
fcl_obj – FCLObject to be added

2. add_object(self: mplib.pymp.PlanningWorld, name: str, collision_object: mplib.pymp.collision_detection.fcl.CollisionObject) -> None

Adds an non-articulated object (`CollisionObjectPtr`) with given name to world

Parameters

- **name** – name of the collision object
- **collision_object** – collision object to be added

add_point_cloud(
self: mplib.pymp.PlanningWorld,
name: str,
vertices: numpy.ndarray[numpy.float64[m, 3]],
resolution: float = 0.01,
) → None

Adds a point cloud as a collision object with given name to world

Parameters

- **name** – name of the point cloud collision object
- **vertices** – point cloud vertices matrix
- **resolution** – resolution of the point in `octomap::OcTree`

attach_box(
self: mplib.pymp.PlanningWorld,
size: numpy.ndarray[numpy.float64[3, 1]],
art_name: str,
link_id: int,
pose: mplib.pymp.Pose,
) → None

Attaches given box to specified link of articulation (auto touch_links)

Parameters

- **size** – box side length
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

attach_mesh(
self: mplib.pymp.PlanningWorld,
mesh_path: str,
art_name: str,
link_id: int,
pose: mplib.pymp.Pose,
*,
convex: bool = False,
) → None

Attaches given mesh to specified link of articulation (auto touch_links)

Parameters

- **mesh_path** – path to a mesh file
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

- **convex** – whether to load mesh as a convex mesh. Default: `False`.

attach_object(**args*, ***kwargs*)

Overloaded function.

1. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int, touch_links: list[str]) -> None

Attaches existing non-articulated object to specified link of articulation at its current pose. If the object is currently attached, disallow collision between the object and previous touch_links.

Updates **acm_** to allow collisions between attached object and touch_links.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **touch_links** – link names that the attached object touches

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

2. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int) -> None

Attaches existing non-articulated object to specified link of articulation at its current pose. If the object is not currently attached, automatically sets touch_links as the name of self links that collide with the object in the current state.

Updates **acm_** to allow collisions between attached object and touch_links.

If the object is already attached, the touch_links of the attached object is preserved and **acm_** remains unchanged.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

3. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int, pose: mplib.pymp.Pose, touch_links: list[str]) -> None

Attaches existing non-articulated object to specified link of articulation at given pose. If the object is currently attached, disallow collision between the object and previous touch_links.

Updates **acm_** to allow collisions between attached object and touch_links.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to

- **pose** – attached pose (relative pose from attached link to object)
- **touch_links** – link names that the attached object touches

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

4. attach_object(self: mplib.pymp.PlanningWorld, name: str, art_name: str, link_id: int, pose: mplib.pymp.Pose) -> None

Attaches existing non-articulated object to specified link of articulation at given pose. If the object is not currently attached, automatically sets touch_links as the name of self links that collide with the object in the current state.

Updates **acm_** to allow collisions between attached object and touch_links.

If the object is already attached, the touch_links of the attached object is preserved and **acm_** remains unchanged.

Parameters

- **name** – name of the non-articulated object to attach
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

Raises

ValueError – if non-articulated object with given name does not exist or if planned articulation with given name does not exist

5. attach_object(self: mplib.pymp.PlanningWorld, name: str, p_geom: mplib.pymp.collision_detection.fcl.CollisionGeometry, art_name: str, link_id: int, pose: mplib.pymp.Pose, touch_links: list[str]) -> None

Attaches given object (w/ p_geom) to specified link of articulation at given pose. This is done by removing the object and then adding and attaching object. As a result, all previous **acm_** entries with the object are removed

Parameters

- **name** – name of the non-articulated object to attach
- **p_geom** – pointer to a CollisionGeometry object
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)
- **touch_links** – link names that the attached object touches

6. attach_object(self: mplib.pymp.PlanningWorld, name: str, p_geom: mplib.pymp.collision_detection.fcl.CollisionGeometry, art_name: str, link_id: int, pose: mplib.pymp.Pose) -> None

Attaches given object (w/ p_geom) to specified link of articulation at given pose. This is done by removing the object and then adding and attaching object. As a result, all previous **acm_** entries with the object are

removed. Automatically sets touch_links as the name of self links that collide with the object in the current state (auto touch_links).

Parameters

- **name** – name of the non-articulated object to attach
- **p_geom** – pointer to a CollisionGeometry object
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

attach_sphere(
self: mplib.pymp.PlanningWorld,
radius: float,
art_name: str,
link_id: int,
pose: mplib.pymp.Pose,
) → None

Attaches given sphere to specified link of articulation (auto touch_links)

Parameters

- **radius** – sphere radius
- **art_name** – name of the planned articulation to attach to
- **link_id** – index of the link of the planned articulation to attach to
- **pose** – attached pose (relative pose from attached link to object)

check_collision(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f18bf0>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check full collision (calls `checkSelfCollision()` and `checkRobotCollision()`)

Parameters
request – collision request params.

Returns
List of `WorldCollisionResult` objects

check_robot_collision(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fd0bd0f70>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check collision with other scene bodies in the world (planned articulations with attached objects collide against unplanned articulations and scene objects)

Parameters
request – collision request params.

Returns
List of `WorldCollisionResult` objects

check_self_collision(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc9b18af0>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check for self collision (including planned articulation self-collision, planned articulation-attach collision, attach-attach collision)

Parameters
request – collision request params.

Returns
List of `WorldCollisionResult` objects

detach_object(
self: mplib.pymp.PlanningWorld,
name: str,
also_remove: bool = False,
) → bool

Detaches object with given name. Updates **acm_** to disallow collision between the object and touch_links.

Parameters

- **name** – name of the non-articulated object to detach
- **also_remove** – whether to also remove object from world

Returns
`True` if success, `False` if the object with given name is not attached

distance(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fc8f125b0>,
) → *mplib.pymp.collision_detection.WorldDistanceResult*

Compute the minimum distance-to-all-collision (calls `distanceSelf()` and `distanceRobot()`)

Parameters
request – distance request params.

Returns
a `WorldDistanceResult` object

distance_robot(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fd0b66270>,
) → *mplib.pymp.collision_detection.WorldDistanceResult*

Compute the minimum distance-to-collision between a robot and the world

Parameters
request – distance request params.

Returns
a `WorldDistanceResult` object

distance_self(
self: mplib.pymp.PlanningWorld,
request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fd0b978b0>,
) → *mplib.pymp.collision_detection.WorldDistanceResult*

Get the minimum distance to self-collision given the robot in current state

Parameters
request – distance request params.

Returns
a `WorldDistanceResult` object

distance_to_collision(*self:* mplib.pymp.PlanningWorld) → float

Compute the minimum distance-to-all-collision. Calls `distance()`.

Note that this is different from MoveIt2's `planning_scene::PlanningScene::distanceToCollision()` where self-collisions are ignored.

Returns
minimum distance-to-all-collision

distance_to_robot_collision(*self:* mplib.pymp.PlanningWorld) → float

The distance between the robot model at current state to the nearest collision (ignoring self-collisions). Calls `distanceRobot()`.

Returns
minimum distance-to-robot-collision

distance_to_self_collision(*self:* mplib.pymp.PlanningWorld) → float

The minimum distance to self-collision given the robot in current state. Calls `distanceSelf()`.

Returns
minimum distance-to-self-collision

get_allowed_collision_matrix(
self: mplib.pymp.PlanningWorld,
) → *mplib.pymp.collision_detection.AllowedCollisionMatrix*

Get the current allowed collision matrix

get_articulation(
self: mplib.pymp.PlanningWorld,
name: str,
) → *mplib.pymp.ArticulatedModel*

Gets the articulation (ArticulatedModelPtr) with given name

Parameters
name – name of the articulated model

Returns
the articulated model with given name or `None` if not found.

get_articulation_names(*self:* mplib.pymp.PlanningWorld) → list[str]

Gets names of all articulations in world (unordered)

get_attached_object(
self: mplib.pymp.PlanningWorld,
name: str,
) → *mplib.pymp.AttachedBody*

Gets the attached body (AttachedBodyPtr) with given name

Parameters
name – name of the attached body

Returns
the attached body with given name or `None` if not found.

`get_object(`
self: mplib.pymp.PlanningWorld,
name: str,
`)` → *mplib.pymp.collision_detection.fcl.FCLObject*

Gets the non-articulated object (`FCLObjectPtr`) with given name

Parameters
name – name of the non-articulated object

Returns
the object with given name or `None` if not found.

`get_object_names`(*self:* mplib.pymp.PlanningWorld) → list[str]

Gets names of all objects in world (unordered)

`get_planned_articulations(`
self: mplib.pymp.PlanningWorld,
`)` → list[*mplib.pymp.ArticulatedModel*]

Gets all planned articulations (ArticulatedModelPtr)

`has_articulation`(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Check whether the articulation with given name exists

Parameters
name – name of the articulated model

Returns
`True` if exists, `False` otherwise.

`has_object`(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Check whether the non-articulated object with given name exists

Parameters
name – name of the non-articulated object

Returns
`True` if exists, `False` otherwise.

`is_articulation_planned(`
self: mplib.pymp.PlanningWorld,
name: str,
`)` → bool

Check whether the articulation with given name is being planned

Parameters
name – name of the articulated model

Returns
`True` if exists, `False` otherwise.

`is_object_attached(`
self: mplib.pymp.PlanningWorld,
name: str,
`)` → bool

Check whether the non-articulated object with given name is attached

Parameters
name – name of the non-articulated object

Returns
`True` if it is attached, `False` otherwise.

is_state_colliding(*self:* mplib.pymp.PlanningWorld) → bool

Check if the current state is in collision (with the environment or self collision).

Returns
`True` if collision exists

print_attached_body_pose(*self:* mplib.pymp.PlanningWorld) → None

Prints global pose of all attached bodies

remove_articulation(
self: mplib.pymp.PlanningWorld,
name: str,
) → bool

Removes the articulation with given name if exists. Updates **acm_**

Parameters
name – name of the articulated model

Returns
`True` if success, `False` if articulation with given name does not exist

remove_object(*self:* mplib.pymp.PlanningWorld, *name: str*) → bool

Removes (and detaches) the collision object with given name if exists. Updates **acm_**

Parameters
name – name of the non-articulated collision object

Returns
`True` if success, `False` if the non-articulated object with given name does not exist

set_articulation_planned(
self: mplib.pymp.PlanningWorld,
name: str,
planned: bool,
) → None

Sets articulation with given name as being planned

Parameters

- **name** – name of the articulated model
- **planned** – whether the articulation is being planned

Raises
ValueError – if the articulation with given name does not exist

set_qpos(
self: mplib.pymp.PlanningWorld,
name: str,
qpos: numpy.ndarray[numpy.float64[m, 1]],
) → None

Set qpos of articulation with given name

Parameters

- **name** – name of the articulated model
- **qpos** – joint angles of the *movegroup only* // FIXME: double check

set_qpos_all(
self: mplib.pymp.PlanningWorld,
state: numpy.ndarray[numpy.float64[m, 1]],
) → None

Set qpos of all planned articulations

class mplib.sapien_utils.SapienPlanner(
sapien_planning_world: SapienPlanningWorld,
move_group: str,
*,
joint_vel_limits: Sequence[float] | ndarray | None = None,
joint_acc_limits: Sequence[float] | ndarray | None = None,
)

Bases: *Planner*

__init__(
sapien_planning_world: SapienPlanningWorld,
move_group: str,
*,
joint_vel_limits: Sequence[float] | ndarray | None = None,
joint_acc_limits: Sequence[float] | ndarray | None = None,
)

Creates an mplib.planner.Planner from a SapienPlanningWorld.

Parameters

- **sapien_planning_world** – a SapienPlanningWorld created from sapien.Scene
- **move_group** – name of the move group (end effector link)
- **joint_vel_limits** – joint velocity limits for time parameterization
- **joint_acc_limits** – joint acceleration limits for time parameterization

update_from_simulation(*, *update_attached_object: bool = True*) → None

Updates PlanningWorld's articulations/objects pose with current Scene state. Note that shape's local_pose is not updated. If those are changed, please recreate a new SapienPlanningWorld instance.

Directly calls `SapienPlanningWorld.update_from_simulation()`

Parameters
update_attached_object – whether to update the attached pose of all attached objects

IK(
goal_pose: Pose,
start_qpos: ndarray,
mask: Sequence[bool] | ndarray | None = None,
*,
n_init_qpos: int = 20,
threshold: float = 1e-3,
return_closest: bool = False,
verbose: bool = False,
) → tuple[str, list[ndarray] | ndarray | None]

Compute inverse kinematics

Parameters

- **goal_pose** – goal pose
- **start_qpos** – initial configuration, (ndof,) np.floating np.ndarray.

- **mask** – qpos mask to disable IK sampling, (ndof,) bool np.ndarray.
- **n_init_qpos** – number of random initial configurations to sample.
- **threshold** – distance threshold for marking sampled IK as success. distance is position error norm + quaternion error norm.
- **return_closest** – whether to return the qpos that is closest to start_qpos, considering equivalent joint values.
- **verbose** – whether to print collision info if any collision exists.

Returns

(status, q_goals)

status: IK status, "Success" if succeeded.

q_goals: list of sampled IK qpos, (ndof,) np.floating np.ndarray.
IK is successful if q_goals is not None. If return_closest, q_goals is np.ndarray if successful and None if not successful.

TOPP(*path*, *step=0.1*, *verbose=False*)

Time-Optimal Path Parameterization

Parameters

- **path** – numpy array of shape (n, dof)
- **step** – step size for the discretization
- **verbose** – if True, will print the log of TOPPRA

check_for_collision(
collision_function,
state: ndarray | None = None,
) → list[*WorldCollisionResult*]

Helper function to check for collision

Parameters
state – all planned articulations qpos state. If None, use current qpos.

Returns
A list of collisions.

check_for_env_collision(
state: ndarray | None = None,
) → list[*WorldCollisionResult*]

Check if the robot is in collision with the environment

Parameters
state – all planned articulations qpos state. If None, use current qpos.

Returns
A list of collisions.

check_for_self_collision(
state: ndarray | None = None,
) → list[*WorldCollisionResult*]

Check if the robot is in self-collision.

Parameters
state – all planned articulations qpos state. If None, use current qpos.

Returns
A list of collisions.

`detach_object`(*name='attached_geom'*, *also_remove=False*) → bool

Detact the attached object with given name

Parameters

- **name** – object name to detach
- **also_remove** – whether to also remove object from world

Returns
`True` if success, `False` if the object with given name is not attached

`pad_move_group_qpos`(*qpos*, *articulation=None*)

If qpos contains only the move_group joints, return qpos padded with current values of the remaining joints of articulation. Otherwise, verify number of joints and return.

Parameters

- **qpos** – joint positions
- **articulation** – the articulation to get qpos from. If None, use self.robot

Returns
joint positions with full dof

`plan_pose`(
goal_pose: Pose,
current_qpos: ndarray,
mask: list[bool] | ndarray | None = None,
*,
time_step: float = 0.1,
rrt_range: float = 0.1,
planning_time: float = 1,
fix_joint_limits: bool = True,
wrt_world: bool = True,
simplify: bool = True,
constraint_function: Callable | None = None,
constraint_jacobian: Callable | None = None,
constraint_tolerance: float = 1e-3,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

plan from a start configuration to a goal pose of the end-effector

Parameters

- **goal_pose** – pose of the goal
- **current_qpos** – current joint configuration (either full or move_group joints)
- **mask** – if the value at a given index is True, the joint is *not* used in the IK
- **time_step** – time step for TOPPRA (time parameterization of path)
- **rrt_range** – step size for RRT
- **planning_time** – time limit for RRT
- **fix_joint_limits** – if True, will clip the joint configuration to be within the joint limits

- **wrt_world** – if true, interpret the target pose with respect to the world frame instead of the base frame
- **verbose** – if True, will print the log of OMPL and TOPPRA

plan_qpos(
goal_qposes: list[ndarray],
current_qpos: ndarray,
***,
time_step: float = 0.1,
rrt_range: float = 0.1,
planning_time: float = 1,
fix_joint_limits: bool = True,
fixed_joint_indices: list[int] | None = None,
simplify: bool = True,
constraint_function: Callable[[ndarray, ndarray], None] | None = None,
constraint_jacobian: Callable[[ndarray, ndarray], None] | None = None,
constraint_tolerance: float = 1e-3,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

Plan a path from a specified joint position to a goal pose

Parameters

- **goal_qposes** – list of target joint configurations, [(ndof,)]
- **current_qpos** – current joint configuration (either full or move_group joints)
- **mask** – mask for IK. When set, the IK will leave certain joints out of planning
- **time_step** – time step for TOPP
- **rrt_range** – step size for RRT
- **planning_time** – time limit for RRT
- **fix_joint_limits** – if True, will clip the joint configuration to be within the joint limits
- **simplify** – whether the planned path will be simplified. (constained planning does not support simplification)
- **constraint_function** – evals to 0 when constraint is satisfied
- **constraint_jacobian** – jacobian of constraint_function
- **constraint_tolerance** – tolerance for constraint_function
- **fixed_joint_indices** – list of indices of joints that are fixed during planning
- **verbose** – if True, will print the log of OMPL and TOPPRA

plan_screw(
goal_pose: Pose,
current_qpos: ndarray,
***,
qpos_step: float = 0.1,
time_step: float = 0.1,
wrt_world: bool = True,
verbose: bool = False,
) → dict[str, str | ndarray | float64]

Plan from a start configuration to a goal pose of the end-effector using screw motion

Parameters

- **goal_pose** – pose of the goal
- **current_qpos** – current joint configuration (either full or move_group joints)
- **qpos_step** – size of the random step
- **time_step** – time step for the discretization
- **wrt_world** – if True, interpret the target pose with respect to the world frame instead of the base frame
- **verbose** – if True, will print the log of TOPPRA

remove_object(*name*) → bool

returns true if the object was removed, false if it was not found

remove_point_cloud(*name='scene_pcd'*) → bool

Removes the point cloud collision object with given name

Parameters
name – name of the point cloud collision object

Returns
`True` if success, `False` if the non-articulation object with given name does not exist

set_base_pose(*pose:* Pose)

tell the planner where the base of the robot is w.r.t the world frame

Parameters
pose – pose of the base

update_attached_box(
size: Sequence[float] | ndarray[tuple[Literal[3], Literal[1]], dtype[floating]],
pose: Pose,
art_name=None,
link_id=-1,
)

Attach a box to some link

Parameters

- **size** – box side length
- **pose** – attaching pose (relative pose from attached link to object)
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_attached_mesh(
mesh_path: str,
pose: Pose,
art_name=None,
link_id=-1,
)

Attach a mesh to some link

Parameters

- **mesh_path** – path to a mesh file
- **pose** – attaching pose (relative pose from attached link to object)

- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_attached_object(
collision_geometry: CollisionGeometry,
pose: Pose,
name='attached_geom',
art_name=None,
link_id=-1,
)

Attach given object (w/ collision geometry) to specified link of articulation

Parameters

- **collision_geometry** – FCL collision geometry
- **pose** – attaching pose (relative pose from attached link to object)
- **name** – name of the attached geometry.
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_attached_sphere(
radius: float,
pose: Pose,
art_name=None,
link_id=-1,
)

Attach a sphere to some link

Parameters

- **radius** – radius of the sphere
- **pose** – attaching pose (relative pose from attached link to object)
- **art_name** – name of the articulated object to attach to. If None, attach to self.robot.
- **link_id** – if not provided, the end effector will be the target.

update_point_cloud(*points*, *resolution=1e-3*, *name='scene_pcd'*)

Adds a point cloud as a collision object with given name to world. If the name is the same, the point cloud is simply updated.

Parameters

- **points** – points, numpy array of shape (n, 3)
- **resolution** – resolution of the point OcTree
- **name** – name of the point cloud collision object

wrap_joint_limit(*qpos: ndarray*) → bool

Checks if the joint configuration can be wrapped to be within the joint limits. For revolute joints, the joint angle is wrapped to be within [q_min, q_min+2*pi)

Parameters

qpos – joint positions, angles of revolute joints might be modified. If not within_limits (returns False), qpos might not be fully wrapped.

Returns
whether qpos can be wrapped to be within the joint limits.

4.1.9 collision_detection

class mplib.collision_detection.**AllowedCollision**

Bases: pybind11_object

AllowedCollision Enum class

Members:

NEVER : Collision is never allowed

ALWAYS : Collision is always allowed

CONDITIONAL : Collision contact is allowed depending on a predicate

ALWAYS = <AllowedCollision.ALWAYS: 1>

CONDITIONAL = <AllowedCollision.CONDITIONAL: 2>

NEVER = <AllowedCollision.NEVER: 0>

__init__(
self: mplib.pymp.collision_detection.AllowedCollision,
value: int,
) → None

property name

property value

class mplib.collision_detection.**AllowedCollisionMatrix**

Bases: pybind11_object

AllowedCollisionMatrix for collision checking

All elements in the collision world are referred to by their names. This class represents which collisions are allowed to happen and which are not.

Mimicking MoveIt2's collision_detection::AllowedCollisionMatrix

https://moveit.picknik.ai/main/api/html/classcollision__detection_1_1AllowedCollisionMatrix.html

__init__(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
) → None

clear(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
) → None

Clear all data in the allowed collision matrix

get_all_entry_names(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
) → list[str]

Get sorted names of all existing elements (including **default_entries_**)

get_allowed_collision(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
name1: str,
name2: str,
) → *mplib.pymp.collision_detection.AllowedCollision* | None

Get the type of the allowed collision between two elements

Returns
AllowedCollision. This is * `None` if the entry does not exist (collision is not allowed) * the entry if an entry or a default entry exists.

get_default_entry(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
name: str,
) → *mplib.pymp.collision_detection.AllowedCollision* | None

Get the default type of the allowed collision for an element

Parameters
name – name of the element

Returns
an AllowedCollision Enum or `None` if the default entry does not exist

get_entry(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
name1: str,
name2: str,
) → *mplib.pymp.collision_detection.AllowedCollision* | None

Get the type of the allowed collision between two elements

Parameters

- **name1** – name of the first element
- **name2** – name of the second element

Returns
an AllowedCollision Enum or `None` if the entry does not exist.

has_default_entry(
self: mplib.pymp.collision_detection.AllowedCollisionMatrix,
name: str,
) → bool

Check if a default entry exists for an element

has_entry(**args*, ***kwargs*)

Overloaded function.

1. has_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str) -> bool

Check if an entry exists for an element

2. has_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name1: str, name2: str) -> bool

Check if an entry exists for a pair of elements

remove_default_entry(**args*, ***kwargs*)

Overloaded function.

1. remove_default_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str) -> None

Remove the default entry for the element if exists

2. remove_default_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, names: list[str]) -> None

Remove the existing default entries for the elements

remove_entry(**args*, ***kwargs*)

Overloaded function.

1. remove_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name1: str, name2: str) -> None

Remove the entry for a pair of elements if exists

2. remove_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str, other_names: list[str]) -> None

Remove existing entries between the element and each element in other_names

3. remove_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, names1: list[str], names2: list[str]) -> None

Remove any existing entries for all possible pairs among two sets of elements

4. remove_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str) -> None

Remove all entries for all possible pairs between the element and existing elements

5. remove_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, names: list[str]) -> None

Remove all entries for all possible pairs between each of the elements and existing elements

set_default_entry(**args*, ***kwargs*)

Overloaded function.

1. set_default_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str, allowed: bool) -> None

Set the default value for entries that include name but are not set explicitly with setEntry(). Apply to future changes of the element set.

2. set_default_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, names: list[str], allowed: bool) -> None

Set the default entries for the elements. Apply to future changes of the element set.

set_entry(**args*, ***kwargs*)

Overloaded function.

1. set_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name1: str, name2: str, allowed: bool) -> None

Set an entry for a pair of elements

2. set_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str, other_names: list[str], allowed: bool) -> None

Set the entries between the element and each element in other_names

3. set_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, names1: list[str], names2: list[str], allowed: bool) -> None

Set the entries for all possible pairs among two sets of elements

4. set_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, name: str, allowed: bool) -> None

Set the entries for all possible pairs between the element and existing elements. As the set of elements might change in the future, consider using setDefaultEntry() instead.

5. set_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, names: list[str], allowed: bool) -> None

Set the entries for all possible pairs between each of the elements and existing elements. As the set of elements might change in the future, consider using setDefaultEntry() instead.

6. set_entry(self: mplib.pymp.collision_detection.AllowedCollisionMatrix, allowed: bool) -> None

Set the entries for all possible pairs among all existing elements

class mplib.collision_detection.**WorldCollisionResult**

Bases: pybind11_object

Result of the collision checking.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.WorldCollisionResult) -> None

Default constructor

2. __init__(self: mplib.pymp.collision_detection.WorldCollisionResult, res: fcl::CollisionResult<double>, collision_type: str, object_name1: str, object_name2: str, link_name1: str, link_name2: str) -> None

Constructor with all members

property collision_type

type of the collision

property link_name1

link name of the first object in collision

property link_name2

link name of the second object in collision

property object_name1

name of the first object

property object_name2

name of the second object

property res

the fcl CollisionResult

class mplib.collision_detection.**WorldDistanceResult**

Bases: pybind11_object

Result of minimum distance-to-collision query.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.WorldDistanceResult) -> None

Default constructor

2. __init__(self: mplib.pymp.collision_detection.WorldDistanceResult, res: fcl::DistanceResult<double>, min_distance: float, distance_type: str, object_name1: str, object_name2: str, link_name1: str, link_name2: str) -> None

Constructor with all members

property distance_type

type of the distance result

property link_name1

link name of the first object

property link_name2

link name of the second object

property min_distance

minimum distance between the two objects

property object_name1

name of the first object

property object_name2

name of the second object

property res

the fcl DistanceResult

fcl

class mplib.collision_detection.fcl.**BVHModel**

Bases: *CollisionGeometry*

BVHModel collision geometry.

Inheriting from CollisionGeometry, this class specializes to a mesh geometry represented by a BVH tree.

__init__(*self:* mplib.pymp.collision_detection.fcl.BVHModel) → None

property aabb_center

property aabb_radius

add_sub_model(**args*, ***kwargs*)

Overloaded function.

1. add_sub_model(self: mplib.pymp.collision_detection.fcl.BVHModel, vertices: list[numpy.ndarray[numpy.float64[3, 1]]]) -> int

Add a sub-model to the BVHModel.

Parameters
vertices – vertices of the sub-model

2. add_sub_model(self: mplib.pymp.collision_detection.fcl.BVHModel, vertices: list[numpy.ndarray[numpy.float64[3, 1]]], faces: list[mplib.pymp.collision_detection.fcl.Triangle]) -> int

Add a sub-model to the BVHModel.

Parameters

- **vertices** – vertices of the sub-model
- **faces** – faces of the sub-model represented by a list of vertex indices

3. add_sub_model(self: mplib.pymp.collision_detection.fcl.BVHModel, vertices: list[numpy.ndarray[numpy.float64[3, 1]]], faces: list[numpy.ndarray[numpy.int32[3, 1]]]) -> None

Add a sub-model to the BVHModel.

Parameters

- **vertices** – vertices of the sub-model
- **faces** – faces of the sub-model represented by a list of vertex indices

begin_model(
self: mplib.pymp.collision_detection.fcl.BVHModel,
num_faces: int = 0,
num_vertices: int = 0,
) → int

Begin to construct a BVHModel.

Parameters

- **num_faces** – number of faces of the mesh
- **num_vertices** – number of vertices of the mesh

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

end_model(*self:* mplib.pymp.collision_detection.fcl.BVHModel) → int

End the construction of a BVHModel.

get_faces(
self: mplib.pymp.collision_detection.fcl.BVHModel,
) → list[*mplib.pymp.collision_detection.fcl.Triangle*]

Get the faces of the BVHModel.

Returns
faces of the BVHModel

get_vertices(
self: mplib.pymp.collision_detection.fcl.BVHModel,
) → list[numpy.ndarray[numpy.float64[3, 1]]]

Get the vertices of the BVHModel.

Returns
vertices of the BVHModel

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

property num_faces

property num_vertices

class mplib.collision_detection.fcl.**Box**

Bases: *CollisionGeometry*

Box collision geometry.

Inheriting from CollisionGeometry, this class specializes to a box geometry.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Box, side: numpy.ndarray[numpy.float64[3, 1]]) -> None

Construct a box with given side length.

Parameters
side – side length of the box in an array [x, y, z]

2. __init__(self: mplib.pymp.collision_detection.fcl.Box, x: float, y: float, z: float) -> None

Construct a box with given side length.

Parameters

- **x** – side length of the box in x direction
- **y** – side length of the box in y direction
- **z** – side length of the box in z direction

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

property side

class mplib.collision_detection.fcl.Capsule

Bases: *CollisionGeometry*

Capsule collision geometry.

Inheriting from CollisionGeometry, this class specializes to a capsule geometry.

__init__(
self: mplib.pymp.collision_detection.fcl.Capsule,
radius: float,
lz: float,
) → None

Construct a capsule with given radius and height.

Parameters

- **radius** – radius of the capsule
- **lz** – height of the capsule along z axis

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

property lz

property radius

class mplib.collision_detection.fcl.**CollisionGeometry**

Bases: pybind11_object

Collision geometry base class.

This is an FCL class so you can refer to the FCL doc here. https://flexible-collision-library.github.io/d6/d5d/classfcl_1_1CollisionGeometry.html

__init__(**args*, ***kwargs*)

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

is_occupied(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

is_uncertain(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

class mplib.collision_detection.fcl.**CollisionObject**

Bases: pybind11_object

Collision object class.

This class contains the collision geometry and the transformation of the geometry.

__init__(
self: mplib.pymp.collision_detection.fcl.CollisionObject,
collision_geometry: mplib.pymp.collision_detection.fcl.CollisionGeometry,
pose: mplib.pymp.Pose = *Pose([0, 0, 0], [1, 0, 0, 0])*,
) → None

Construct a collision object with given collision geometry and transformation.

Parameters

- **collision_geometry** – collision geometry of the object
- **pose** – pose of the object

get_collision_geometry(
self: mplib.pymp.collision_detection.fcl.CollisionObject,
) → *mplib.pymp.collision_detection.fcl.CollisionGeometry*

get_pose(
self: mplib.pymp.collision_detection.fcl.CollisionObject,
) → *mplib.pymp.Pose*

Gets the current pose of the collision object in world

Returns
The current pose of the collision object

property pose

Pose of the collision object in world

set_pose(
self: mplib.pymp.collision_detection.fcl.CollisionObject,
pose: mplib.pymp.Pose,
) → None

Sets the pose of the collision object in world

Parameters
pose – New pose of the collision object

class mplib.collision_detection.fcl.**CollisionRequest**

Bases: pybind11_object

__init__(
self: mplib.pymp.collision_detection.fcl.CollisionRequest,
num_max_contacts: int = 1,
enable_contact: bool = False,
num_max_cost_sources: int = 1,

enable_cost: bool = False,
use_approximate_cost: bool = True,
gjk_solver_type: mplib.pymp.collision_detection.fcl.GJKSolverType = <GJKSolverType.GST_LIBCCD: 0>,
gjk_tolerance: float = 1e-06,
) → None

is_satisfied(
self: mplib.pymp.collision_detection.fcl.CollisionRequest,
result: fcl::CollisionResult<double>,
) → bool

class mplib.collision_detection.fcl.**CollisionResult**

Bases: pybind11_object

__init__(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
) → None

add_contact(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
c: fcl::Contact<double>,
) → None

add_cost_source(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
c: fcl::CostSource<double>,
num_max_cost_sources: int,
) → None

clear(*self:* mplib.pymp.collision_detection.fcl.CollisionResult) → None

get_contact(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
i: int,
) → fcl::Contact<double>

get_contacts(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
) → list[fcl::Contact<double>]

get_cost_sources(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
) → list[fcl::CostSource<double>]

is_collision(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
) → bool

num_contacts(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
) → int

num_cost_sources(
self: mplib.pymp.collision_detection.fcl.CollisionResult,
) → int

class mplib.collision_detection.fcl.**Cone**

Bases: *CollisionGeometry*

Cone collision geometry.

Inheriting from CollisionGeometry, this class specializes to a cone geometry.

__init__(
self: mplib.pymp.collision_detection.fcl.Cone,
radius: float,
lz: float,
) → None

Construct a cone with given radius and height.

Parameters

- **radius** – radius of the cone
- **lz** – height of the cone along z axis

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

property lz

property radius

class mplib.collision_detection.fcl.**Contact**

Bases: pybind11_object

__init__(*args, **kwargs)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Contact) -> None
2. __init__(self: mplib.pymp.collision_detection.fcl.Contact, o1: mplib.pymp.collision_detection.fcl.CollisionGeometry, o2: mplib.pymp.collision_detection.fcl.CollisionGeometry, b1: int, b2: int) -> None
3. __init__(self: mplib.pymp.collision_detection.fcl.Contact, o1: mplib.pymp.collision_detection.fcl.CollisionGeometry, o2: mplib.pymp.collision_detection.fcl.CollisionGeometry, b1: int, b2: int, pos: numpy.ndarray[numpy.float64[3, 1]], normal: numpy.ndarray[numpy.float64[3, 1]], depth: float) -> None

property normal

property penetration_depth

property pos

class mplib.collision_detection.fcl.ContactPoint

Bases: pybind11_object

__init__(*args, **kwargs)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.ContactPoint) -> None
2. __init__(self: mplib.pymp.collision_detection.fcl.ContactPoint, normal: numpy.ndarray[numpy.float64[3, 1]], pos: numpy.ndarray[numpy.float64[3, 1]], penetration_depth: float) -> None

property normal

property penetration_depth

property pos

class mplib.collision_detection.fcl.Convex

Bases: *CollisionGeometry*

Convex collision geometry.

Inheriting from CollisionGeometry, this class specializes to a convex geometry.

__init__(*args, **kwargs)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Convex, vertices: std::vector<Eigen::Matrix<double, 3, 1, 0, 3, 1>, std::allocator<Eigen::Matrix<double, 3, 1, 0, 3, 1> > >, num_faces: int, faces: std::vector<int, std::allocator<int> >, throw_if_invalid: bool = True) -> None

Construct a convex with given vertices and faces.

Parameters

- **vertices** – vertices of the convex
- **num_faces** – number of faces of the convex
- **faces** – faces of the convex geometry represented by a list of vertex indices
- **throw_if_invalid** – if True, throw an exception if the convex is invalid

2. __init__(self: mplib.pymp.collision_detection.fcl.Convex, vertices: numpy.ndarray[numpy.float64[m, 3]], faces: numpy.ndarray[numpy.int32[m, 3]], throw_if_invalid: bool = True) -> None

Construct a convex with given vertices and faces.

Parameters

- **vertices** – vertices of the convex
- **faces** – faces of the convex geometry represented by a list of vertex indices
- **throw_if_invalid** – if `True`, throw an exception if the convex is invalid

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(*self:* mplib.pymp.collision_detection.fcl.Convex) → float

Compute the volume of the convex.

Returns
volume of the convex

property cost_density

get_face_count(*self:* mplib.pymp.collision_detection.fcl.Convex) → int

Get the number of faces of the convex.

Returns
number of faces of the convex

get_faces(*self:* mplib.pymp.collision_detection.fcl.Convex) → list[int]

Get the faces of the convex.

Returns
faces of the convex represented by a list of vertex indices

get_interior_point(
self: mplib.pymp.collision_detection.fcl.Convex,
) → numpy.ndarray[numpy.float64[3, 1]]

Sample a random interior point of the convex geometry

Returns
interior point of the convex

get_vertices(
self: mplib.pymp.collision_detection.fcl.Convex,
) → list[numpy.ndarray[numpy.float64[3, 1]]]

Get the vertices of the convex.

Returns
vertices of the convex

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

class mplib.collision_detection.fcl.**CostSource**

Bases: pybind11_object

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.CostSource) -> None
2. __init__(self: mplib.pymp.collision_detection.fcl.CostSource, aabb_min: numpy.ndarray[numpy.float64[3, 1]], aabb_max: numpy.ndarray[numpy.float64[3, 1]], cost_density: float) -> None

property aabb_max

property aabb_min

property cost_density

property total_cost

class mplib.collision_detection.fcl.**Cylinder**

Bases: *CollisionGeometry*

Cylinder collision geometry.

Inheriting from CollisionGeometry, this class specializes to a cylinder geometry.

__init__(
self: mplib.pymp.collision_detection.fcl.Cylinder,
radius: float,
lz: float,
) → None

Construct a cylinder with given radius and height.

Parameters

- **radius** – radius of the cylinder
- **lz** – height of the cylinder along z axis

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

property lz

property radius

class mplib.collision_detection.fcl.**DistanceRequest**

Bases: pybind11_object

__init__(
self: mplib.pymp.collision_detection.fcl.DistanceRequest,
enable_nearest_points: bool = False,
enable_signed_distance: bool = False,
rel_err: float = 0.0,
abs_err: float = 0.0,
distance_tolerance: float = 1e-06,
gjk_solver_type: mplib.pymp.collision_detection.fcl.GJKSolverType = <GJKSolverType.GST_LIBCCD: 0>,
) → None

is_satisfied(
self: mplib.pymp.collision_detection.fcl.DistanceRequest,
result: fcl::DistanceResult<double>,
) → bool

class mplib.collision_detection.fcl.**DistanceResult**

Bases: pybind11_object

__init__(
self: mplib.pymp.collision_detection.fcl.DistanceResult,
min_distance: float = 1.7976931348623157e+308,
) → None

clear(*self:* mplib.pymp.collision_detection.fcl.DistanceResult) → None

property min_distance

property nearest_points

class mplib.collision_detection.fcl.**Ellipsoid**

Bases: *CollisionGeometry*

Ellipsoid collision geometry.

Inheriting from CollisionGeometry, this class specializes to a ellipsoid geometry.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Ellipsoid, a: float, b: float, c: float) -> None

Construct a ellipsoid with given parameters.

Parameters

- **a** – length of the x semi-axis
- **b** – length of the y semi-axis
- **c** – length of the z semi-axis

2. __init__(self: mplib.pymp.collision_detection.fcl.Ellipsoid, radii: numpy.ndarray[numpy.float64[3, 1]]) -> None

Construct a ellipsoid with given parameters.

Parameters
radii – vector of the length of the x, y, and z semi-axes

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

is_uncertain(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

property radii

class mplib.collision_detection.fcl.**FCLModel**

Bases: pybind11_object

FCL collision model of an articulation

See https://github.com/flexible-collision-library/fcl

__init__(
self: mplib.pymp.collision_detection.fcl.FCLModel,
urdf_filename: str,
*,
convex: bool = False,
verbose: bool = False,
) → None

Construct an FCL model from URDF and SRDF files.

Parameters

- **urdf_filename** – path to URDF file, can be relative to the current working directory
- **convex** – use convex decomposition for collision objects. Default: `False`.
- **verbose** – print debug information. Default: `False`.

check_collision_with(**args*, ***kwargs*)

Overloaded function.

1. check_collision_with(self: mplib.pymp.collision_detection.fcl.FCLModel, other: mplib.pymp.collision_detection.fcl.FCLModel, request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f128b0>, *, acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc8f12870>) -> list[mplib.pymp.collision_detection.WorldCollisionResult]

Check for collision in the current state with another `FCLModel`, ignoring the distances between links that are allowed to always collide (as specified by acm).

Parameters

- **other** – another `FCLModel` to check collision with
- **acm** – allowed collision matrix.
- **request** – collision request

Returns

List of `WorldCollisionResult` objects. If empty, no collision.

2. check_collision_with(self: mplib.pymp.collision_detection.fcl.FCLModel, object: mplib::collision_detection::fcl::FCLObject<double>, request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f03d70>, *, acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc8f0cc30>) -> list[mplib.pymp.collision_detection.WorldCollisionResult]

Check for collision in the current state with an `FCLObject`, ignoring the distances between objects that are allowed to always collide (as specified by acm).

Parameters

- **object** – an `FCLObject` to check collision with
- **acm** – allowed collision matrix.
- **request** – collision request

Returns

List of `WorldCollisionResult` objects. If empty, no collision.

check_self_collision(
self: mplib.pymp.collision_detection.fcl.FCLModel,
request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f126f0>,
*,
acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc9b949f0>,
) → list[*mplib.pymp.collision_detection.WorldCollisionResult*]

Check for self-collision in the current state and returns all found collisions, ignoring the distances between links that are allowed to always collide (as specified by acm).

Parameters

- **request** – collision request
- **acm** – allowed collision matrix.

Returns

List of `WorldCollisionResult` objects. If empty, no self-collision.

static create_from_urdf_string(*urdf_string: str*, *collision_links: list[mplib::collision_detection::fcl::FCLObject<double>]*, *, *verbose: bool = False*) → *mplib.pymp.collision_detection.fcl.FCLModel*

Constructs a FCLModel from URDF string and collision links

Parameters

- **urdf_string** – URDF string (without visual/collision elements for links)
- **collision_links** – Vector of collision links as FCLObjectPtr. Format is: `[FCLObjectPtr, ...]`. The collision objects are at the shape's local_pose.
- **verbose** – print debug information. Default: `False`.

Returns

a unique_ptr to FCLModel

distance_self(
self: mplib.pymp.collision_detection.fcl.FCLModel,

request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fd3da12f0>,
*,
acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fd0bc9db0>,
) → *mplib.pymp.collision_detection.WorldDistanceResult*

Get the minimum distance to self-collision given the robot in current state, ignoring the distances between links that are allowed to always collide (as specified by acm).

Parameters

- **request** – distance request.
- **acm** – allowed collision matrix.

Returns
a `WorldDistanceResult` object

distance_to_collision_with(**args*, ***kwargs*)

Overloaded function.

1. distance_to_collision_with(self: mplib.pymp.collision_detection.fcl.FCLModel, other: mplib.pymp.collision_detection.fcl.FCLModel, *, acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc8f12df0>) -> float

The minimum distance to collision with another `FCLModel` given the robot in current state, ignoring the distances between links that are allowed to always collide (as specified by acm).

Parameters

- **other** – another `FCLModel` to get minimum distance-to-collision with
- **acm** – allowed collision matrix.

Returns
minimum distance-to-collision with the other `FCLModel`

2. distance_to_collision_with(self: mplib.pymp.collision_detection.fcl.FCLModel, object: mplib::collision_detection::fcl::FCLObject<double>, *, acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fd3970cf0>) -> float

The minimum distance to collision with an `FCLObject` given the robot in current state, ignoring the distances between objects that are allowed to always collide (as specified by acm).

Parameters

- **object** – an `FCLObject` to get minimum distance-to-collision with
- **acm** – allowed collision matrix.

Returns
minimum distance-to-collision with the `FCLObject`

distance_to_self_collision(
self: mplib.pymp.collision_detection.fcl.FCLModel,
*,
acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fd0b7ab30>,
) → float

The minimum distance to self-collision given the robot in current state, ignoring the distances between links that are allowed to always collide (as specified by acm). Calls `distanceSelf()`.

Parameters
acm – allowed collision matrix.

Returns
minimum distance-to-self-collision

distance_with(**args*, ***kwargs*)

Overloaded function.

1. distance_with(self: mplib.pymp.collision_detection.fcl.FCLModel, other: mplib.pymp.collision_detection.fcl.FCLModel, request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fc9f64a70>, *, acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc8efb5b0>) -> mplib.pymp.collision_detection.WorldDistanceResult

Get the minimum distance to collision with another `FCLModel` given the robot in current state, ignoring the distances between links that are allowed to always collide (as specified by acm).

Parameters

- **other** – another `FCLModel` to get minimum distance-to-collision with
- **request** – distance request.
- **acm** – allowed collision matrix.

Returns
a `WorldDistanceResult` object

2. distance_with(self: mplib.pymp.collision_detection.fcl.FCLModel, object: mplib::collision_detection::fcl::FCLObject<double>, request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fc8f131f0>, *, acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc8f03230>) -> mplib.pymp.collision_detection.WorldDistanceResult

Get the minimum distance to collision with an `FCLObject` given the robot in current state, ignoring the distances between objects that are allowed to always collide (as specified by acm).

Parameters

- **object** – an `FCLObject` to get minimum distance-to-collision with
- **request** – distance request.
- **acm** – allowed collision matrix.

Returns
a `WorldDistanceResult` object

get_collision_link_names(
self: mplib.pymp.collision_detection.fcl.FCLModel,
) → list[str]

get_collision_objects(
self: mplib.pymp.collision_detection.fcl.FCLModel,
) → list[mplib::collision_detection::fcl::FCLObject<double>]

Get the collision objects of the FCL model.

Returns
all collision objects of the FCL model

get_collision_pairs(
self: mplib.pymp.collision_detection.fcl.FCLModel,
) → list[tuple[int, int]]

Get the collision pairs of the FCL model.

Returns
collision pairs of the FCL model. If the FCL model has N collision objects, the collision pairs is a list of N*(N-1)/2 pairs minus the disabled collision pairs

get_name(*self:* mplib.pymp.collision_detection.fcl.FCLModel) → str

Get name of the articulated model.

Returns
name of the articulated model

is_state_colliding(
self: mplib.pymp.collision_detection.fcl.FCLModel,
*,
acm: mplib.pymp.collision_detection.AllowedCollisionMatrix = <mplib.pymp.collision_detection.AllowedCollisionMatrix object at 0x7f6fc9f650b0>,
) → bool

Check if the current state is in self-collision, ignoring the distances between links that are allowed to always collide (as specified by acm).

Parameters
acm – allowed collision matrix.

Returns
`True` if any collision pair collides and `False` otherwise.

property name

Name of the fcl model

remove_collision_pairs_from_srdf(
self: mplib.pymp.collision_detection.fcl.FCLModel,
srdf_filename: str,
) → None

Remove collision pairs from SRDF file.

Parameters
srdf_filename – path to SRDF file, can be relative to the current working directory

set_link_order(
self: mplib.pymp.collision_detection.fcl.FCLModel,
names: list[str],
) → None

Set the link order of the FCL model.

Parameters
names – list of link names in the order that you want to set.

update_collision_objects(
self: mplib.pymp.collision_detection.fcl.FCLModel,
*link_poses: list[*mplib.pymp.Pose*]*,
) → None

Update the collision objects of the FCL model.

Parameters
link_poses – list of link poses in the order of the link order

class mplib.collision_detection.fcl.**FCLObject**

Bases: pybind11_object

A general high-level object which consists of multiple FCLCollisionObjects. It is the top level data structure which is used in the collision checking process.

Mimicking MoveIt2's collision_detection::FCLObject and collision_detection::World::Object

https://moveit.picknik.ai/main/api/html/structcollision__detection_1_1FCLObject.html https://moveit.picknik.ai/main/api/html/structcollision__detection_1_1World_1_1Object.html

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.FCLObject, name: str) -> None

Construct a new FCLObject with the given name

Parameters
name – name of this FCLObject

2. __init__(self: mplib.pymp.collision_detection.fcl.FCLObject, name: str, pose: mplib.pymp.Pose, shapes: list[mplib.pymp.collision_detection.fcl.CollisionObject], shape_poses: list[mplib.pymp.Pose]) -> None

Construct a new FCLObject with the given name and shapes

Parameters

- **name** – name of this FCLObject
- **pose** – pose of this FCLObject. All shapes are relative to this pose
- **shapes** – all collision shapes as a vector of fcl::CollisionObjectPtr
- **shape_poses** – relative poses from this FCLObject to each collision shape

property name

Name of this FCLObject

property pose

Pose of this FCLObject. All shapes are relative to this pose

property shape_poses

Relative poses from this FCLObject to each collision shape

property shapes

All collision shapes (fcl::CollisionObjectPtr) making up this FCLObject

class mplib.collision_detection.fcl.**GJKSolverType**

Bases: pybind11_object

Members:

GST_LIBCCD

GST_INDEP

GST_INDEP = <GJKSolverType.GST_INDEP: 1>

GST_LIBCCD = <GJKSolverType.GST_LIBCCD: 0>

__init__(
self: mplib.pymp.collision_detection.fcl.GJKSolverType,
value: int,
) → None

property name

property value

class mplib.collision_detection.fcl.**Halfspace**

Bases: *CollisionGeometry*

Infinite halfspace collision geometry.

Inheriting from CollisionGeometry, this class specializes to a halfspace geometry.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Halfspace, n: numpy.ndarray[numpy.float64[3, 1]], d: float) -> None

Construct a halfspace with given normal direction and offset where `n * p = d`. Points in the negative side of the separation plane `{p | n * p < d}` are inside the half space (will have collision).

Parameters

- **n** – normal direction of the halfspace
- **d** – offset of the halfspace

2. __init__(self: mplib.pymp.collision_detection.fcl.Halfspace, a: float, b: float, c: float, d: float) -> None

Construct a halfspace with given halfspace parameters where `ax + by + cz = d`. Points in the negative side of the separation plane `{(x, y, z) | ax + by + cz < d}` are inside the half space (will have collision).

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

property d

distance(
self: mplib.pymp.collision_detection.fcl.Halfspace,
p: numpy.ndarray[numpy.float64[3, 1]],
) → float

Compute the distance of a point to the halfspace as `abs(n * p - d)`.

Parameters
p – a point in 3D space

Returns
distance of the point to the halfspace

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

is_uncertain(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

property n

signed_distance(
self: mplib.pymp.collision_detection.fcl.Halfspace,
p: numpy.ndarray[numpy.float64[3, 1]],
) → float

Compute the signed distance of a point to the halfspace as `n * p - d`.

Parameters
p – a point in 3D space

Returns
signed distance of the point to the halfspace

class mplib.collision_detection.fcl.**OcTree**

Bases: *CollisionGeometry*

OcTree collision geometry.

Inheriting from CollisionGeometry, this class specializes to a point cloud geometry represented by an OcTree.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.OcTree, resolution: float = 0.01) -> None

Construct an OcTree with given resolution.

Parameters
resolution – resolution of the OcTree (smallest size of a voxel). You can treat this is as the diameter of a point. Default is 0.01.

2. __init__(self: mplib.pymp.collision_detection.fcl.OcTree, vertices: numpy.ndarray[numpy.float64[m, 3]], resolution: float = 0.01) -> None

Construct an OcTree with given vertices and resolution.

Parameters

- **vertices** – vertices of the point cloud
- **resolution** – resolution of the OcTree. Default is 0.01

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

class mplib.collision_detection.fcl.**Plane**

Bases: *CollisionGeometry*

Infinite plane collision geometry.

Inheriting from CollisionGeometry, this class specializes to a plane geometry.

__init__(**args*, ***kwargs*)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Plane, n: numpy.ndarray[numpy.float64[3, 1]], d: float) -> None

Construct a plane with given normal direction and offset where `n * p = d`.

Parameters

- **n** – normal direction of the plane
- **d** – offset of the plane

2. __init__(self: mplib.pymp.collision_detection.fcl.Plane, a: float, b: float, c: float, d: float) -> None

Construct a plane with given plane parameters where `ax + by + cz = d`.

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

property d

distance(
self: mplib.pymp.collision_detection.fcl.Plane,
p: numpy.ndarray[numpy.float64[3, 1]],
) → float

Compute the distance of a point to the plane as `abs(n * p - d)`.

Parameters
p – a point in 3D space

Returns
distance of the point to the plane

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

property n

signed_distance(
self: mplib.pymp.collision_detection.fcl.Plane,
p: numpy.ndarray[numpy.float64[3, 1]],
) → float

Compute the signed distance of a point to the plane as `n * p - d`.

Parameters
p – a point in 3D space

Returns
signed distance of the point to the plane

class mplib.collision_detection.fcl.**Sphere**

Bases: *CollisionGeometry*

Sphere collision geometry.

Inheriting from CollisionGeometry, this class specializes to a sphere geometry.

__init__(*self:* mplib.pymp.collision_detection.fcl.Sphere, *radius: float*) → None

Construct a sphere with given radius.

Parameters
radius – radius of the sphere

property aabb_center

property aabb_radius

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_uncertain(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

property radius

class mplib.collision_detection.fcl.Triangle

Bases: pybind11_object

Triangle with 3 indices for points.

This is an FCL class so you can refer to the FCL doc here. https://flexible-collision-library.github.io/de/daa/classfcl_1_1Triangle.html

__init__(*args, **kwargs)

Overloaded function.

1. __init__(self: mplib.pymp.collision_detection.fcl.Triangle) -> None
2. __init__(self: mplib.pymp.collision_detection.fcl.Triangle, p1: int, p2: int, p3: int) -> None

get(*self:* mplib.pymp.collision_detection.fcl.Triangle, *arg0: int*) → int

set(
self: mplib.pymp.collision_detection.fcl.Triangle,
arg0: int,
arg1: int,
arg2: int,
) → None

class mplib.collision_detection.fcl.TriangleP

Bases: *CollisionGeometry*

TriangleP collision geometry.

Inheriting from CollisionGeometry, this class specializes to a triangleP geometry.

__init__(
self: mplib.pymp.collision_detection.fcl.TriangleP,
a: numpy.ndarray[numpy.float64[3, 1]],
b: numpy.ndarray[numpy.float64[3, 1]],
c: numpy.ndarray[numpy.float64[3, 1]],
) → None

Construct a set of triangles from vectors of point coordinates.

Parameters

- **a** – vector of point x coordinates
- **b** – vector of point y coordinates
- **c** – vector of point z coordinates

property a

property aabb_center

property aabb_radius

property b

property c

compute_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 1]]

compute_local_aabb(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → None

compute_moment_of_inertia(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_moment_of_inertia_related_to_com(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → numpy.ndarray[numpy.float64[3, 3]]

compute_volume(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → float

property cost_density

is_free(*self:* mplib.pymp.collision_detection.fcl.CollisionGeometry) → bool

is_occupied(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

is_uncertain(
self: mplib.pymp.collision_detection.fcl.CollisionGeometry,
) → bool

mplib.collision_detection.fcl.**collide**(**args*, ***kwargs*)

Overloaded function.

1. collide(obj1: mplib.pymp.collision_detection.fcl.CollisionObject, obj2: mplib.pymp.collision_detection.fcl.CollisionObject, request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f01b30>) -> mplib.pymp.collision_detection.fcl.CollisionResult

2. collide(obj1: mplib.pymp.collision_detection.fcl.FCLObject, obj2: mplib.pymp.collision_detection.fcl.FCLObject, request: mplib.pymp.collision_detection.fcl.CollisionRequest = <mplib.pymp.collision_detection.fcl.CollisionRequest object at 0x7f6fc8f0f3f0>) -> mplib.pymp.collision_detection.fcl.CollisionResult

mplib.collision_detection.fcl.**distance**(**args*, ***kwargs*)

Overloaded function.

1. distance(obj1: mplib.pymp.collision_detection.fcl.CollisionObject, obj2: mplib.pymp.collision_detection.fcl.CollisionObject, request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fc8f120b0>) -> mplib.pymp.collision_detection.fcl.DistanceResult

2. distance(obj1: mplib.pymp.collision_detection.fcl.FCLObject, obj2: mplib.pymp.collision_detection.fcl.FCLObject, request: mplib.pymp.collision_detection.fcl.DistanceRequest = <mplib.pymp.collision_detection.fcl.DistanceRequest object at 0x7f6fc8efb130>) -> mplib.pymp.collision_detection.fcl.DistanceResult

mplib.collision_detection.fcl.**load_mesh_as_BVH**(
mesh_path: str,
scale: numpy.ndarray[numpy.float64[3, 1]],
) → *mplib.pymp.collision_detection.fcl.BVHModel*

Load a triangle mesh from mesh_path as a non-convex collision object.

Parameters

- **mesh_path** – path to the mesh
- **scale** – mesh scale factor

Returns
a shared_ptr to an fcl::BVHModel_OBBRSS<S> collision object

mplib.collision_detection.fcl.**load_mesh_as_Convex**(
mesh_path: str,
scale: numpy.ndarray[numpy.float64[3, 1]],
) → *mplib.pymp.collision_detection.fcl.Convex*

Load a convex mesh from mesh_path.

Parameters

- **mesh_path** – path to the mesh
- **scale** – mesh scale factor

Returns
a shared_ptr to an fcl::Convex<S> collision object

Raises
RuntimeError – if the mesh is not convex.

4.1.10 kinematics

Kinematics submodule

pinocchio

class mplib.kinematics.pinocchio.**PinocchioModel**

Bases: pybind11_object

Pinocchio model of an articulation

See https://github.com/stack-of-tasks/pinocchio

__init__(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
urdf_filename: str,
*,
gravity: numpy.ndarray[numpy.float64[3, 1]] = array([0., 0., -9.81]),
verbose: bool = False,
) → None

Construct a Pinocchio model from the given URDF file.

Parameters

- **urdf_filename** – path to the URDF file
- **gravity** – gravity vector, by default is `[0, 0, -9.81]` in -z axis
- **verbose** – print debug information. Default: `False`.

compute_IK_CLIK(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
pose: mplib.pymp.Pose,
q_init: numpy.ndarray[numpy.float64[m, 1]],

mask: list[bool] = [],
eps: float = 1e-05,
max_iter: int = 1000,
dt: float = 0.1,
damp: float = 1e-12,
) → tuple[numpy.ndarray[numpy.float64[m, 1]], bool, numpy.ndarray[numpy.float64[6, 1]]]

Compute the inverse kinematics using close loop inverse kinematics.

Parameters

- **index** – index of the link (in the order you passed to the constructor or the default order)
- **pose** – desired pose of the link
- **q_init** – initial joint configuration
- **mask** – if the value at a given index is `True`, the joint is *not* used in the IK
- **eps** – tolerance for the IK
- **max_iter** – maximum number of iterations
- **dt** – time step for the CLIK
- **damp** – damping for the CLIK

Returns
joint configuration

compute_IK_CLIK_JL(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
pose: mplib.pymp.Pose,
q_init: numpy.ndarray[numpy.float64[m, 1]],
q_min: numpy.ndarray[numpy.float64[m, 1]],
q_max: numpy.ndarray[numpy.float64[m, 1]],
eps: float = 1e-05,
max_iter: int = 1000,
dt: float = 0.1,
damp: float = 1e-12,
) → tuple[numpy.ndarray[numpy.float64[m, 1]], bool, numpy.ndarray[numpy.float64[6, 1]]]

The same as `compute_IK_CLIK()` but with it clamps the joint configuration to the given limits.

Parameters

- **index** – index of the link (in the order you passed to the constructor or the default order)
- **pose** – desired pose of the link
- **q_init** – initial joint configuration
- **q_min** – minimum joint configuration
- **q_max** – maximum joint configuration
- **eps** – tolerance for the IK
- **max_iter** – maximum number of iterations
- **dt** – time step for the CLIK
- **damp** – damping for the CLIK

Returns
joint configuration

compute_forward_kinematics(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
qpos: numpy.ndarray[numpy.float64[m, 1]],
) → None

Compute forward kinematics for the given joint configuration.

If you want the result you need to call `get_link_pose()`

Parameters
qpos – joint configuration. Needs to be full configuration, not just the movegroup joints.

compute_full_jacobian(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
qpos: numpy.ndarray[numpy.float64[m, 1]],
) → None

Compute the full jacobian for the given joint configuration. Note you need to call computeForwardKinematics() first. If you want the result you need to call `get_link_jacobian()`

Parameters
qpos – joint configuration. Needs to be full configuration, not just the movegroup joints.

compute_single_link_jacobian(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
qpos: numpy.ndarray[numpy.float64[m, 1]],
index: int,
local: bool = False,
) → numpy.ndarray[numpy.float64[6, n]]

Compute the jacobian of the given link. Note you need to call computeForwardKinematics() first.

Parameters

- **qpos** – joint configuration. Needs to be full configuration, not just the movegroup joints.
- **index** – index of the link (in the order you passed to the constructor or the default order)
- **local** – if `True` return the jacobian w.r.t. the instantaneous local frame of the link

Returns
6 x n jacobian of the link

static create_from_urdf_string(
urdf_string: str,
*,
gravity: numpy.ndarray[numpy.float64[3, 1]] = array([0., 0., -9.81]),
verbose: bool = False,
) → *mplib.pymp.kinematics.pinocchio.PinocchioModel*

Constructs a PinocchioModel from URDF string

Parameters

- **urdf_string** – URDF string (without visual/collision elements for links)
- **gravity** – gravity vector, by default is `[0, 0, -9.81]` in -z axis
- **verbose** – print debug information. Default: `False`.

Returns
a unique_ptr to PinocchioModel

get_adjacent_links(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
) → set[tuple[str, str]]

Get the all adjacent link names.

Returns
adjacent link names as a set of pairs of strings

get_chain_joint_index(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
end_effector: str,
) → list[int]

Get the joint indices of the joints in the chain from the root to the given link.

Parameters
index – index of the link (in the order you passed to the constructor or the default order)

Returns
joint indices of the joints in the chain

get_chain_joint_name(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
end_effector: str,
) → list[str]

Get the joint names of the joints in the chain from the root to the given link.

Parameters
index – index of the link (in the order you passed to the constructor or the default order)

Returns
joint names of the joints in the chain

get_joint_dim(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
user: bool = True,
) → int

Get the dimension of the joint with the given index.

Parameters

- **index** – joint index to query
- **user** – if `True`, the joint index follows the order you passed to the constructor or the default order

Returns
dimension of the joint with the given index

get_joint_dims(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
) → numpy.ndarray[numpy.int32[m, 1]]

Get the dimension of all the joints. Again, Pinocchio might split a joint into multiple joints.

Parameters
user – if `True`, we get the dimension of the joints in the order you passed to the constructor or the default order

Returns
dimention of the joints

`get_joint_id(`
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
user: bool = True,
`) →` int

Get the id of the joint with the given index.

Parameters

- **index** – joint index to query
- **user** – if `True`, the joint index follows the order you passed to the constructor or the default order

Returns
id of the joint with the given index

`get_joint_ids(`
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
`) →` numpy.ndarray[numpy.int32[m, 1]]

Get the id of all the joints. Again, Pinocchio might split a joint into multiple joints.

Parameters
user – if `True`, we get the id of the joints in the order you passed to the constructor or the default order

Returns
id of the joints

`get_joint_limit(`
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
user: bool = True,
`) →` numpy.ndarray[numpy.float64[m, n]]

Get the limit of the joint with the given index.

Parameters

- **index** – joint index to query
- **user** – if `True`, the joint index follows the order you passed to the constructor or the default order

Returns
limit of the joint with the given index

`get_joint_limits(`
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
`) →` list[numpy.ndarray[numpy.float64[m, n]]]

Get the limit of all the joints. Again, Pinocchio might split a joint into multiple joints.

Parameters
user – if `True`, we get the limit of the joints in the order you passed to the constructor or the default order

Returns
limit of the joints

get_joint_names(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
) → list[str]

Get the name of all the joints. Again, Pinocchio might split a joint into multiple joints.

Parameters
user – if True, we get the name of the joints in the order you passed to the constructor or the default order

Returns
name of the joints

get_joint_parent(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
user: bool = True,
) → int

Get the parent of the joint with the given index.

Parameters

- **index** – joint index to query
- **user** – if True, the joint index follows the order you passed to the constructor or the default order

Returns
parent of the joint with the given index

get_joint_parents(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
) → numpy.ndarray[numpy.int32[m, 1]]

Get the parent of all the joints. Again, Pinocchio might split a joint into multiple joints.

Parameters
user – if True, we get the parent of the joints in the order you passed to the constructor or the default order

Returns
parent of the joints

get_joint_type(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
user: bool = True,
) → str

Get the type of the joint with the given index.

Parameters

- **index** – joint index to query
- **user** – if True, the joint index follows the order you passed to the constructor or the default order

Returns
type of the joint with the given index

get_joint_types(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
) → list[str]

Get the type of all the joints. Again, Pinocchio might split a joint into multiple joints.

Parameters
user – if `True`, we get the type of the joints in the order you passed to the constructor or the default order

Returns
type of the joints

get_leaf_links(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
) → list[str]

Get the leaf links (links without child) of the kinematic tree.

Returns
list of leaf links

get_link_jacobian(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
local: bool = False,
) → numpy.ndarray[numpy.float64[6, n]]

Get the jacobian of the given link. You must call `compute_full_jacobian()` first.

Parameters

- **index** – index of the link (in the order you passed to the constructor or the default order)
- **local** – if `True`, the jacobian is expressed in the local frame of the link, otherwise it is expressed in the world frame

Returns
6 x n jacobian of the link

get_link_names(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
user: bool = True,
) → list[str]

Get the name of all the links.

Parameters
user – if `True`, we get the name of the links in the order you passed to the constructor or the default order

Returns
name of the links

get_link_pose(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
index: int,
) → *mplib.pymp.Pose*

Get the pose of the given link in robot's base (root) frame.

Parameters
index – index of the link (in the order you passed to the constructor or the default order)

Returns
pose of the link in robot's base (root) frame.

get_random_configuration(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
) → numpy.ndarray[numpy.float64[m, 1]]

Get a random configuration.

Returns
random joint configuration

print_frames(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
) → None

Frame is a Pinocchio internal data type which is not supported outside this class.

set_joint_order(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
names: list[str],
) → None

Pinocchio might have a different joint order or it might add additional joints.

If you do not pass the the list of joint names, the default order might not be the one you want.

Parameters
names – list of joint names in the order you want

set_link_order(
self: mplib.pymp.kinematics.pinocchio.PinocchioModel,
names: list[str],
) → None

Pinocchio might have a different link order or it might add additional links.

If you do not pass the the list of link names, the default order might not be the one you want.

Parameters
names – list of link names in the order you want

kdl

class mplib.kinematics.kdl.**KDLModel**

Bases: pybind11_object

KDL model of an articulation

See https://github.com/orocos/orocos_kinematics_dynamics

__init__(
self: mplib.pymp.kinematics.kdl.KDLModel,
urdf_filename: str,
link_names: list[str],
joint_names: list[str],
*,
verbose: bool = False,
) → None

chain_IK_LMA(
self: mplib.pymp.kinematics.kdl.KDLModel,
index: int,
q_init: numpy.ndarray[numpy.float64[m, 1]],
goal_pose: mplib.pymp.Pose,
) → tuple[numpy.ndarray[numpy.float64[m, 1]], int]

chain_IK_NR(
self: mplib.pymp.kinematics.kdl.KDLModel,
index: int,
q_init: numpy.ndarray[numpy.float64[m, 1]],
goal_pose: mplib.pymp.Pose,
) → tuple[numpy.ndarray[numpy.float64[m, 1]], int]

chain_IK_NR_JL(
self: mplib.pymp.kinematics.kdl.KDLModel,
index: int,
q_init: numpy.ndarray[numpy.float64[m, 1]],
goal_pose: mplib.pymp.Pose,
q_min: numpy.ndarray[numpy.float64[m, 1]],
q_max: numpy.ndarray[numpy.float64[m, 1]],
) → tuple[numpy.ndarray[numpy.float64[m, 1]], int]

get_tree_root_name(*self:* mplib.pymp.kinematics.kdl.KDLModel) → str

tree_IK_NR_JL(
self: mplib.pymp.kinematics.kdl.KDLModel,
endpoints: list[str],
q_init: numpy.ndarray[numpy.float64[m, 1]],
*goal_poses: list[*mplib.pymp.Pose*]*,
q_min: numpy.ndarray[numpy.float64[m, 1]],
q_max: numpy.ndarray[numpy.float64[m, 1]],
) → tuple[numpy.ndarray[numpy.float64[m, 1]], int]

4.1.11 planning

Planning submodule

ompl

class mplib.planning.ompl.**FixedJoint**

Bases: pybind11_object

__init__(
self: mplib.pymp.planning.ompl.FixedJoint,
articulation_idx: int,
joint_idx: int,
value: float,
) → None

property articulation_idx

property joint_idx

property value

class mplib.planning.ompl.**OMPLPlanner**

Bases: pybind11_object

OMPL Planner

__init__(
self: mplib.pymp.planning.ompl.OMPLPlanner,
world: mplib::PlanningWorldTpl<double>,
) → None

Construct an OMPLPlanner from a PlanningWorld

Parameters
world – planning world

plan(*self: mplib.pymp.planning.ompl.OMPLPlanner, start_state: numpy.ndarray[numpy.float64[m, 1]], goal_states: list[numpy.ndarray[numpy.float64[m, 1]]], *, time: float = 1.0, range: float = 0.0, fixed_joints: set[mplib::planning::ompl::FixedJointTpl<double>] = set(), simplify: bool = True, constraint_function: Callable[[numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1], flags.writeable]], None] = None, constraint_jacobian: Callable[[numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1], flags.writeable]], None] = None, constraint_tolerance: float = 0.001, verbose: bool = False*) → tuple[str, numpy.ndarray[numpy.float64[m, n]]]

Plan a path from start state to goal states.

Parameters

- **start_state** – start state of the movegroup joints
- **goal_states** – list of goal states. Planner will stop when one of them is reached
- **time** – planning time limit
- **range** – planning range (for RRT family of planners and represents the maximum step size)
- **fixed_joints** – list of fixed joints not considered in planning for this particular call
- **simplify** – whether the path will be simplified by calling `_simplifyPath()` (constained planning does not support simplification)
- **constraint_function** – a R^d to R^1 function that evals to 0 when constraint is satisfied. Constraint ignored if fixed joints not empty
- **constraint_jacobian** – the jacobian of the constraint w.r.t. the joint angles
- **constraint_tolerance** – tolerance of what level of deviation from 0 is acceptable
- **verbose** – print debug information. Default: `False`.

Returns
pair of planner status and path. If planner succeeds, status is "Exact solution."

simplify_path(
self: mplib.pymp.planning.ompl.OMPLPlanner,
path: numpy.ndarray[numpy.float64[m, n]],
) → numpy.ndarray[numpy.float64[m, n]]

Simplify the provided path.

Parameters
path – path to be simplified (numpy array of shape (n, dim))

Returns
simplified path

4.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

mplib.collision_detection, 61
mplib.collision_detection.fcl, 65
mplib.kinematics.kdl, 98
mplib.kinematics.pinocchio, 91
mplib.planning.ompl, 99
mplib.urdf_utils, 44

INDEX

Symbols

__init__() (*mplib.ArticulatedModel method*), 37
__init__() (*mplib.AttachedBody method*), 40
__init__() (*mplib.Planner method*), 21
__init__() (*mplib.PlanningWorld method*), 28
__init__() (*mplib.Pose method*), 42
__init__() (*mplib.collision_detection.AllowedCollision method*), 61
__init__() (*mplib.collision_detection.AllowedCollisionMatrix method*), 61
__init__() (*mplib.collision_detection.WorldCollisionResult method*), 64
__init__() (*mplib.collision_detection.WorldDistanceResult method*), 64
__init__() (*mplib.collision_detection.fcl.BVHModel method*), 65
__init__() (*mplib.collision_detection.fcl.Box method*), 67
__init__() (*mplib.collision_detection.fcl.Capsule method*), 68
__init__() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
__init__() (*mplib.collision_detection.fcl.CollisionObject method*), 70
__init__() (*mplib.collision_detection.fcl.CollisionRequest method*), 70
__init__() (*mplib.collision_detection.fcl.CollisionResult method*), 71
__init__() (*mplib.collision_detection.fcl.Cone method*), 72
__init__() (*mplib.collision_detection.fcl.Contact method*), 72
__init__() (*mplib.collision_detection.fcl.ContactPoint method*), 73
__init__() (*mplib.collision_detection.fcl.Convex method*), 73
__init__() (*mplib.collision_detection.fcl.CostSource method*), 75
__init__() (*mplib.collision_detection.fcl.Cylinder method*), 75
__init__() (*mplib.collision_detection.fcl.DistanceRequest method*), 76
__init__() (*mplib.collision_detection.fcl.DistanceResult method*), 76
__init__() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
__init__() (*mplib.collision_detection.fcl.FCLModel method*), 78
__init__() (*mplib.collision_detection.fcl.FCLObject method*), 83
__init__() (*mplib.collision_detection.fcl.GJKSolverType method*), 84
__init__() (*mplib.collision_detection.fcl.Halfspace method*), 84
__init__() (*mplib.collision_detection.fcl.OcTree method*), 85
__init__() (*mplib.collision_detection.fcl.Plane method*), 86
__init__() (*mplib.collision_detection.fcl.Sphere method*), 88
__init__() (*mplib.collision_detection.fcl.Triangle method*), 89
__init__() (*mplib.collision_detection.fcl.TriangleP method*), 89
__init__() (*mplib.kinematics.kdl.KDLModel method*), 98
__init__() (*mplib.kinematics.pinocchio.PinocchioModel method*), 91
__init__() (*mplib.planning.ompl.FixedJoint method*), 99
__init__() (*mplib.planning.ompl.OMPLPlanner method*), 100
__init__() (*mplib.sapien_utils.SapienPlanner method*), 55
__init__() (*mplib.sapien_utils.SapienPlanningWorld method*), 45

A

a (*mplib.collision_detection.fcl.TriangleP property*), 89
aabb_center (*mplib.collision_detection.fcl.Box property*), 67
aabb_center (*mplib.collision_detection.fcl.BVHModel property*), 65
aabb_center (*mplib.collision_detection.fcl.Capsule*

property), 68
aabb_center (mplib.collision_detection.fcl.CollisionGeometry property), 69
aabb_center (mplib.collision_detection.fcl.Cone property), 72
aabb_center (mplib.collision_detection.fcl.Convex property), 74
aabb_center (mplib.collision_detection.fcl.Cylinder property), 75
aabb_center (mplib.collision_detection.fcl.Ellipsoid property), 77
aabb_center (mplib.collision_detection.fcl.Halfspace property), 84
aabb_center (mplib.collision_detection.fcl.OcTree property), 86
aabb_center (mplib.collision_detection.fcl.Plane property), 87
aabb_center (mplib.collision_detection.fcl.Sphere property), 88
aabb_center (mplib.collision_detection.fcl.TriangleP property), 89
aabb_max (mplib.collision_detection.fcl.CostSource property), 75
aabb_min (mplib.collision_detection.fcl.CostSource property), 75
aabb_radius (mplib.collision_detection.fcl.Box property), 67
aabb_radius (mplib.collision_detection.fcl.BVHModel property), 65
aabb_radius (mplib.collision_detection.fcl.Capsule property), 68
aabb_radius (mplib.collision_detection.fcl.CollisionGeometry property), 69
aabb_radius (mplib.collision_detection.fcl.Cone property), 72
aabb_radius (mplib.collision_detection.fcl.Convex property), 74
aabb_radius (mplib.collision_detection.fcl.Cylinder property), 75
aabb_radius (mplib.collision_detection.fcl.Ellipsoid property), 77
aabb_radius (mplib.collision_detection.fcl.Halfspace property), 84
aabb_radius (mplib.collision_detection.fcl.OcTree property), 86
aabb_radius (mplib.collision_detection.fcl.Plane property), 87
aabb_radius (mplib.collision_detection.fcl.Sphere property), 88
aabb_radius (mplib.collision_detection.fcl.TriangleP property), 89
add_articulation() (mplib.PlanningWorld method), 28
add_articulation() (mplib.sapien_utils.SapienPlanningWorld method), 46
add_contact() (mplib.collision_detection.fcl.CollisionResult method), 71
add_cost_source() (mplib.collision_detection.fcl.CollisionResult method), 71
add_object() (mplib.PlanningWorld method), 28
add_object() (mplib.sapien_utils.SapienPlanningWorld method), 46
add_point_cloud() (mplib.PlanningWorld method), 28
add_point_cloud() (mplib.sapien_utils.SapienPlanningWorld method), 47
add_sub_model() (mplib.collision_detection.fcl.BVHModel method), 65
AllowedCollision (class in mplib.collision_detection), 61
AllowedCollisionMatrix (class in mplib.collision_detection), 61
ALWAYS (mplib.collision_detection.AllowedCollision attribute), 61
ArticulatedModel (class in mplib), 37
articulation_idx (mplib.planning.ompl.FixedJoint property), 99
attach_box() (mplib.PlanningWorld method), 29
attach_box() (mplib.sapien_utils.SapienPlanningWorld method), 47
attach_mesh() (mplib.PlanningWorld method), 29
attach_mesh() (mplib.sapien_utils.SapienPlanningWorld method), 47
attach_object() (mplib.PlanningWorld method), 29
attach_object() (mplib.sapien_utils.SapienPlanningWorld method), 48
attach_sphere() (mplib.PlanningWorld method), 32
attach_sphere() (mplib.sapien_utils.SapienPlanningWorld method), 50
AttachedBody (class in mplib), 40

B

b (mplib.collision_detection.fcl.TriangleP property), 89
base_pose (mplib.ArticulatedModel property), 37
begin_model() (mplib.collision_detection.fcl.BVHModel method), 66
Box (class in mplib.collision_detection.fcl), 67
BVHModel (class in mplib.collision_detection.fcl), 65

C

c (mplib.collision_detection.fcl.TriangleP property), 89
Capsule (class in mplib.collision_detection.fcl), 68
chain_IK_LMA() (mplib.kinematics.kdl.KDLModel method), 98
chain_IK_NR() (mplib.kinematics.kdl.KDLModel method), 99
chain_IK_NR_JL() (mplib.kinematics.kdl.KDLModel method), 99
check_collision() (mplib.PlanningWorld method), 32

check_collision() (*mplib.sapien_utils.SapienPlanningWorld method*), 50
check_collision_between() (*mplib.sapien_utils.SapienPlanningWorld method*), 45
check_collision_with() (*mplib.collision_detection.fcl.FCLModel method*), 78
check_for_collision() (*mplib.Planner method*), 22
check_for_collision() (*mplib.sapien_utils.SapienPlanner method*), 56
check_for_env_collision() (*mplib.Planner method*), 23
check_for_env_collision() (*mplib.sapien_utils.SapienPlanner method*), 56
check_for_self_collision() (*mplib.Planner method*), 23
check_for_self_collision() (*mplib.sapien_utils.SapienPlanner method*), 56
check_robot_collision() (*mplib.PlanningWorld method*), 32
check_robot_collision() (*mplib.sapien_utils.SapienPlanningWorld method*), 50
check_self_collision() (*mplib.collision_detection.fcl.FCLModel method*), 79
check_self_collision() (*mplib.PlanningWorld method*), 32
check_self_collision() (*mplib.sapien_utils.SapienPlanningWorld method*), 50
clear() (*mplib.collision_detection.AllowedCollisionMatrix method*), 61
clear() (*mplib.collision_detection.fcl.CollisionResult method*), 71
clear() (*mplib.collision_detection.fcl.DistanceResult method*), 76
collide() (*in module mplib.collision_detection.fcl*), 90
collision_type (*mplib.collision_detection.WorldCollisionResult property*), 64
CollisionGeometry (*class in mplib.collision_detection.fcl*), 69
CollisionObject (*class in mplib.collision_detection.fcl*), 70
CollisionRequest (*class in mplib.collision_detection.fcl*), 70
CollisionResult (*class in mplib.collision_detection.fcl*), 71
compute_com() (*mplib.collision_detection.fcl.Box method*), 67
compute_com() (*mplib.collision_detection.fcl.BVHModel method*), 66
compute_com() (*mplib.collision_detection.fcl.Capsule method*), 68
compute_com() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
compute_com() (*mplib.collision_detection.fcl.Cone method*), 72
compute_com() (*mplib.collision_detection.fcl.Convex method*), 74
compute_com() (*mplib.collision_detection.fcl.Cylinder method*), 75
compute_com() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
compute_com() (*mplib.collision_detection.fcl.Halfspace method*), 84
compute_com() (*mplib.collision_detection.fcl.OcTree method*), 86
compute_com() (*mplib.collision_detection.fcl.Plane method*), 87
compute_com() (*mplib.collision_detection.fcl.Sphere method*), 88
compute_com() (*mplib.collision_detection.fcl.TriangleP method*), 89
compute_default_collisions() (*in module mplib.urdf_utils*), 44
compute_forward_kinematics() (*mplib.kinematics.pinocchio.PinocchioModel method*), 93
compute_full_jacobian() (*mplib.kinematics.pinocchio.PinocchioModel method*), 93
compute_IK_CLIK() (*mplib.kinematics.pinocchio.PinocchioModel method*), 91
compute_IK_CLIK_JL() (*mplib.kinematics.pinocchio.PinocchioModel method*), 92
compute_local_aabb() (*mplib.collision_detection.fcl.Box method*), 67
compute_local_aabb() (*mplib.collision_detection.fcl.BVHModel method*), 66
compute_local_aabb() (*mplib.collision_detection.fcl.Capsule method*), 68
compute_local_aabb() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
compute_local_aabb() (*mplib.collision_detection.fcl.Cone method*), 72
compute_local_aabb() (*mplib.collision_detection.fcl.Convex method*), 74
compute_local_aabb() (*mplib.collision_detection.fcl.Cylinder*

method), 75
compute_local_aabb() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
compute_local_aabb() (*mplib.collision_detection.fcl.Halfspace method*), 84
compute_local_aabb() (*mplib.collision_detection.fcl.OcTree method*), 86
compute_local_aabb() (*mplib.collision_detection.fcl.Plane method*), 87
compute_local_aabb() (*mplib.collision_detection.fcl.Sphere method*), 88
compute_local_aabb() (*mplib.collision_detection.fcl.TriangleP method*), 89
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Box method*), 68
compute_moment_of_inertia() (*mplib.collision_detection.fcl.BVHModel method*), 66
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Capsule method*), 68
compute_moment_of_inertia() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Cone method*), 72
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Convex method*), 74
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Cylinder method*), 76
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Halfspace method*), 84
compute_moment_of_inertia() (*mplib.collision_detection.fcl.OcTree method*), 86
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Plane method*), 87
compute_moment_of_inertia() (*mplib.collision_detection.fcl.Sphere method*), 88
compute_moment_of_inertia() (*mplib.collision_detection.fcl.TriangleP method*), 90
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Box method*), 68
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.BVHModel method*), 66
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Capsule method*), 68
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Cone method*), 72
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Convex method*), 74
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Cylinder method*), 76
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Halfspace method*), 84
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.OcTree method*), 86
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Plane method*), 87
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.Sphere method*), 88
compute_moment_of_inertia_related_to_com() (*mplib.collision_detection.fcl.TriangleP method*), 90
compute_single_link_jacobian() (*mplib.kinematics.pinocchio.PinocchioModel method*), 93
compute_volume() (*mplib.collision_detection.fcl.Box method*), 68
compute_volume() (*mplib.collision_detection.fcl.BVHModel method*), 66
compute_volume() (*mplib.collision_detection.fcl.Capsule method*), 69
compute_volume() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
compute_volume() (*mplib.collision_detection.fcl.Cone*

method), 72
compute_volume() (*mplib.collision_detection.fcl.Convex method*), 74
compute_volume() (*mplib.collision_detection.fcl.Cylinder method*), 76
compute_volume() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
compute_volume() (*mplib.collision_detection.fcl.Halfspace method*), 85
compute_volume() (*mplib.collision_detection.fcl.OcTree method*), 86
compute_volume() (*mplib.collision_detection.fcl.Plane method*), 87
compute_volume() (*mplib.collision_detection.fcl.Sphere method*), 88
compute_volume() (*mplib.collision_detection.fcl.TriangleP method*), 90
CONDITIONAL (*mplib.collision_detection.AllowedCollision attribute*), 61
Cone (*class in mplib.collision_detection.fcl*), 71
Contact (*class in mplib.collision_detection.fcl*), 72
ContactPoint (*class in mplib.collision_detection.fcl*), 73
convert_physx_component() (*mplib.sapien_utils.SapienPlanningWorld static method*), 46
Convex (*class in mplib.collision_detection.fcl*), 73
cost_density (*mplib.collision_detection.fcl.Box property*), 68
cost_density (*mplib.collision_detection.fcl.BVHModel property*), 66
cost_density (*mplib.collision_detection.fcl.Capsule property*), 69
cost_density (*mplib.collision_detection.fcl.CollisionGeometry property*), 69
cost_density (*mplib.collision_detection.fcl.Cone property*), 72
cost_density (*mplib.collision_detection.fcl.Convex property*), 74
cost_density (*mplib.collision_detection.fcl.CostSource property*), 75
cost_density (*mplib.collision_detection.fcl.Cylinder property*), 76
cost_density (*mplib.collision_detection.fcl.Ellipsoid property*), 77
cost_density (*mplib.collision_detection.fcl.Halfspace property*), 85
cost_density (*mplib.collision_detection.fcl.OcTree property*), 86
cost_density (*mplib.collision_detection.fcl.Plane property*), 87
cost_density (*mplib.collision_detection.fcl.Sphere property*), 88
cost_density (*mplib.collision_detection.fcl.TriangleP property*), 90
CostSource (*class in mplib.collision_detection.fcl*), 75
create_from_urdf_string() (*mplib.ArticulatedModel static method*), 37
create_from_urdf_string() (*mplib.collision_detection.fcl.FCLModel static method*), 79
create_from_urdf_string() (*mplib.kinematics.pinocchio.PinocchioModel static method*), 93
Cylinder (*class in mplib.collision_detection.fcl*), 75

D

d (*mplib.collision_detection.fcl.Halfspace property*), 85
d (*mplib.collision_detection.fcl.Plane property*), 87
detach_object() (*mplib.Planner method*), 25
detach_object() (*mplib.PlanningWorld method*), 33
detach_object() (*mplib.sapien_utils.SapienPlanner method*), 57
detach_object() (*mplib.sapien_utils.SapienPlanningWorld method*), 51
distance() (*in module mplib.collision_detection.fcl*), 90
distance() (*mplib.collision_detection.fcl.Halfspace method*), 85
distance() (*mplib.collision_detection.fcl.Plane method*), 87
distance() (*mplib.PlanningWorld method*), 33
distance() (*mplib.Pose method*), 42
distance() (*mplib.sapien_utils.SapienPlanningWorld method*), 51
distance_between() (*mplib.sapien_utils.SapienPlanningWorld method*), 45
distance_robot() (*mplib.PlanningWorld method*), 33
distance_robot() (*mplib.sapien_utils.SapienPlanningWorld method*), 51
distance_self() (*mplib.collision_detection.fcl.FCLModel method*), 79
distance_self() (*mplib.PlanningWorld method*), 33
distance_self() (*mplib.sapien_utils.SapienPlanningWorld method*), 51
distance_to_collision() (*mplib.PlanningWorld method*), 33
distance_to_collision() (*mplib.sapien_utils.SapienPlanningWorld method*), 52
distance_to_collision_with() (*mplib.collision_detection.fcl.FCLModel method*), 80
distance_to_robot_collision() (*mplib.PlanningWorld method*), 34
distance_to_robot_collision() (*mplib.sapien_utils.SapienPlanningWorld method*), 52

distance_to_self_collision() (*mplib.collision_detection.fcl.FCLModel method*), 80
distance_to_self_collision() (*mplib.PlanningWorld method*), 34
distance_to_self_collision() (*mplib.sapien_utils.SapienPlanningWorld method*), 52
distance_type (*mplib.collision_detection.WorldDistanceResult property*), 65
distance_with() (*mplib.collision_detection.fcl.FCLModel method*), 81
DistanceRequest (*class in mplib.collision_detection.fcl*), 76
DistanceResult (*class in mplib.collision_detection.fcl*), 76

E

Ellipsoid (*class in mplib.collision_detection.fcl*), 77
end_model() (*mplib.collision_detection.fcl.BVHModel method*), 66

F

FCLModel (*class in mplib.collision_detection.fcl*), 78
FCLObject (*class in mplib.collision_detection.fcl*), 83
FixedJoint (*class in mplib.planning.ompl*), 99

G

generate_srdf() (*in module mplib.urdf_utils*), 44
get() (*mplib.collision_detection.fcl.Triangle method*), 89
get_adjacent_links() (*mplib.kinematics.pinocchio.PinocchioModel method*), 93
get_all_entry_names() (*mplib.collision_detection.AllowedCollisionMatrix method*), 61
get_allowed_collision() (*mplib.collision_detection.AllowedCollisionMatrix method*), 61
get_allowed_collision_matrix() (*mplib.PlanningWorld method*), 34
get_allowed_collision_matrix() (*mplib.sapien_utils.SapienPlanningWorld method*), 52
get_articulation() (*mplib.PlanningWorld method*), 34
get_articulation() (*mplib.sapien_utils.SapienPlanningWorld method*), 52
get_articulation_names() (*mplib.PlanningWorld method*), 34
get_articulation_names() (*mplib.sapien_utils.SapienPlanningWorld method*), 52
get_attached_articulation() (*mplib.AttachedBody method*), 41
get_attached_link_global_pose() (*mplib.AttachedBody method*), 41
get_attached_link_id() (*mplib.AttachedBody method*), 41
get_attached_object() (*mplib.PlanningWorld method*), 34
get_attached_object() (*mplib.sapien_utils.SapienPlanningWorld method*), 52
get_base_pose() (*mplib.ArticulatedModel method*), 38
get_chain_joint_index() (*mplib.kinematics.pinocchio.PinocchioModel method*), 94
get_chain_joint_name() (*mplib.kinematics.pinocchio.PinocchioModel method*), 94
get_collision_geometry() (*mplib.collision_detection.fcl.CollisionObject method*), 70
get_collision_link_names() (*mplib.collision_detection.fcl.FCLModel method*), 81
get_collision_objects() (*mplib.collision_detection.fcl.FCLModel method*), 81
get_collision_pairs() (*mplib.collision_detection.fcl.FCLModel method*), 82
get_contact() (*mplib.collision_detection.fcl.CollisionResult method*), 71
get_contacts() (*mplib.collision_detection.fcl.CollisionResult method*), 71
get_cost_sources() (*mplib.collision_detection.fcl.CollisionResult method*), 71
get_default_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 62
get_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 62
get_face_count() (*mplib.collision_detection.fcl.Convex method*), 74
get_faces() (*mplib.collision_detection.fcl.BVHModel method*), 66
get_faces() (*mplib.collision_detection.fcl.Convex method*), 74
get_fcl_model() (*mplib.ArticulatedModel method*), 38
get_global_pose() (*mplib.AttachedBody method*), 41
get_interior_point() (*mplib.collision_detection.fcl.Convex method*), 74
get_joint_dim() (*mplib.kinematics.pinocchio.PinocchioModel method*), 94

get_joint_dims() (*mplib.kinematics.pinocchio.PinocchioModel method*), 94
get_joint_id() (*mplib.kinematics.pinocchio.PinocchioModel method*), 95
get_joint_ids() (*mplib.kinematics.pinocchio.PinocchioModel method*), 95
get_joint_limit() (*mplib.kinematics.pinocchio.PinocchioModel method*), 95
get_joint_limits() (*mplib.kinematics.pinocchio.PinocchioModel method*), 95
get_joint_names() (*mplib.kinematics.pinocchio.PinocchioModel method*), 96
get_joint_parent() (*mplib.kinematics.pinocchio.PinocchioModel method*), 96
get_joint_parents() (*mplib.kinematics.pinocchio.PinocchioModel method*), 96
get_joint_type() (*mplib.kinematics.pinocchio.PinocchioModel method*), 96
get_joint_types() (*mplib.kinematics.pinocchio.PinocchioModel method*), 97
get_leaf_links() (*mplib.kinematics.pinocchio.PinocchioModel method*), 97
get_link_jacobian() (*mplib.kinematics.pinocchio.PinocchioModel method*), 97
get_link_names() (*mplib.kinematics.pinocchio.PinocchioModel method*), 97
get_link_pose() (*mplib.kinematics.pinocchio.PinocchioModel method*), 97
get_move_group_end_effectors() (*mplib.ArticulatedModel method*), 38
get_move_group_joint_indices() (*mplib.ArticulatedModel method*), 38
get_move_group_joint_names() (*mplib.ArticulatedModel method*), 38
get_move_group_qpos_dim() (*mplib.ArticulatedModel method*), 38
get_name() (*mplib.ArticulatedModel method*), 38
get_name() (*mplib.AttachedBody method*), 41
get_name() (*mplib.collision_detection.fcl.FCLModel method*), 82
get_object() (*mplib.AttachedBody method*), 41
get_object() (*mplib.PlanningWorld method*), 34
get_object() (*mplib.sapien_utils.SapienPlanningWorld method*), 53
get_object_names() (*mplib.PlanningWorld method*), 35
get_object_names() (*mplib.sapien_utils.SapienPlanningWorld method*), 53
get_p() (*mplib.Pose method*), 43
get_pinocchio_model() (*mplib.ArticulatedModel method*), 39
get_planned_articulations() (*mplib.PlanningWorld method*), 35
get_planned_articulations() (*mplib.sapien_utils.SapienPlanningWorld method*), 53
get_pose() (*mplib.ArticulatedModel method*), 39
get_pose() (*mplib.AttachedBody method*), 41
get_pose() (*mplib.collision_detection.fcl.CollisionObject method*), 70
get_q() (*mplib.Pose method*), 43
get_qpos() (*mplib.ArticulatedModel method*), 39
get_random_configuration() (*mplib.kinematics.pinocchio.PinocchioModel method*), 98
get_touch_links() (*mplib.AttachedBody method*), 41
get_tree_root_name() (*mplib.kinematics.kdl.KDLModel method*), 99
get_user_joint_names() (*mplib.ArticulatedModel method*), 39
get_user_link_names() (*mplib.ArticulatedModel method*), 39
get_vertices() (*mplib.collision_detection.fcl.BVHModel method*), 67
get_vertices() (*mplib.collision_detection.fcl.Convex method*), 74
GJKSolverType (*class in mplib.collision_detection.fcl*), 83
GST_INDEP (*mplib.collision_detection.fcl.GJKSolverType attribute*), 83
GST_LIBCCD (*mplib.collision_detection.fcl.GJKSolverType attribute*), 84

H

Halfspace (*class in mplib.collision_detection.fcl*), 84
has_articulation() (*mplib.PlanningWorld method*), 35
has_articulation() (*mplib.sapien_utils.SapienPlanningWorld method*), 53
has_default_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 62
has_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 62
has_object() (*mplib.PlanningWorld method*), 35
has_object() (*mplib.sapien_utils.SapienPlanningWorld method*), 53

I

IK() (*mplib.Planner method*), 23
IK() (*mplib.sapien_utils.SapienPlanner method*), 55
inv() (*mplib.Pose method*), 43
is_articulation_planned() (*mplib.PlanningWorld method*), 35

is_articulation_planned() (*mplib.sapien_utils.SapienPlanningWorld method*), 53
is_collision() (*mplib.collision_detection.fcl.CollisionResult method*), 71
is_free() (*mplib.collision_detection.fcl.Box method*), 68
is_free() (*mplib.collision_detection.fcl.BVHModel method*), 67
is_free() (*mplib.collision_detection.fcl.Capsule method*), 69
is_free() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
is_free() (*mplib.collision_detection.fcl.Cone method*), 72
is_free() (*mplib.collision_detection.fcl.Convex method*), 75
is_free() (*mplib.collision_detection.fcl.Cylinder method*), 76
is_free() (*mplib.collision_detection.fcl.Ellipsoid method*), 77
is_free() (*mplib.collision_detection.fcl.Halfspace method*), 85
is_free() (*mplib.collision_detection.fcl.OcTree method*), 86
is_free() (*mplib.collision_detection.fcl.Plane method*), 87
is_free() (*mplib.collision_detection.fcl.Sphere method*), 88
is_free() (*mplib.collision_detection.fcl.TriangleP method*), 90
is_object_attached() (*mplib.PlanningWorld method*), 35
is_object_attached() (*mplib.sapien_utils.SapienPlanningWorld method*), 53
is_occupied() (*mplib.collision_detection.fcl.Box method*), 68
is_occupied() (*mplib.collision_detection.fcl.BVHModel method*), 67
is_occupied() (*mplib.collision_detection.fcl.Capsule method*), 69
is_occupied() (*mplib.collision_detection.fcl.CollisionGeometry method*), 69
is_occupied() (*mplib.collision_detection.fcl.Cone method*), 72
is_occupied() (*mplib.collision_detection.fcl.Convex method*), 75
is_occupied() (*mplib.collision_detection.fcl.Cylinder method*), 76
is_occupied() (*mplib.collision_detection.fcl.Ellipsoid method*), 78
is_occupied() (*mplib.collision_detection.fcl.Halfspace method*), 85
is_occupied() (*mplib.collision_detection.fcl.OcTree method*), 86
is_occupied() (*mplib.collision_detection.fcl.Plane method*), 87
is_occupied() (*mplib.collision_detection.fcl.Sphere method*), 88
is_occupied() (*mplib.collision_detection.fcl.TriangleP method*), 90
is_satisfied() (*mplib.collision_detection.fcl.CollisionRequest method*), 71
is_satisfied() (*mplib.collision_detection.fcl.DistanceRequest method*), 76
is_state_colliding() (*mplib.collision_detection.fcl.FCLModel method*), 82
is_state_colliding() (*mplib.PlanningWorld method*), 35
is_state_colliding() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
is_uncertain() (*mplib.collision_detection.fcl.Box method*), 68
is_uncertain() (*mplib.collision_detection.fcl.BVHModel method*), 67
is_uncertain() (*mplib.collision_detection.fcl.Capsule method*), 69
is_uncertain() (*mplib.collision_detection.fcl.CollisionGeometry method*), 70
is_uncertain() (*mplib.collision_detection.fcl.Cone method*), 72
is_uncertain() (*mplib.collision_detection.fcl.Convex method*), 75
is_uncertain() (*mplib.collision_detection.fcl.Cylinder method*), 76
is_uncertain() (*mplib.collision_detection.fcl.Ellipsoid method*), 78
is_uncertain() (*mplib.collision_detection.fcl.Halfspace method*), 85
is_uncertain() (*mplib.collision_detection.fcl.OcTree method*), 86
is_uncertain() (*mplib.collision_detection.fcl.Plane method*), 87
is_uncertain() (*mplib.collision_detection.fcl.Sphere method*), 88
is_uncertain() (*mplib.collision_detection.fcl.TriangleP method*), 90

J

joint_idx (*mplib.planning.ompl.FixedJoint property*), 99

K

KDLModel (*class in mplib.kinematics.kdl*), 98

L

link_name1 (*mplib.collision_detection.WorldCollisionResult property*), 64
link_name1 (*mplib.collision_detection.WorldDistanceResult property*), 65
link_name2 (*mplib.collision_detection.WorldCollisionResult property*), 64
link_name2 (*mplib.collision_detection.WorldDistanceResult property*), 65
load_mesh_as_BVH() (*in module mplib.collision_detection.fcl*), 90
load_mesh_as_Convex() (*in module mplib.collision_detection.fcl*), 91
lz (*mplib.collision_detection.fcl.Capsule property*), 69
lz (*mplib.collision_detection.fcl.Cone property*), 72
lz (*mplib.collision_detection.fcl.Cylinder property*), 76

M

min_distance (*mplib.collision_detection.fcl.DistanceResult property*), 77
min_distance (*mplib.collision_detection.WorldDistanceResult property*), 65
module
    mplib.collision_detection, 61
    mplib.collision_detection.fcl, 65
    mplib.kinematics.kdl, 98
    mplib.kinematics.pinocchio, 91
    mplib.planning.ompl, 99
    mplib.urdf_utils, 44
mplib.collision_detection
    module, 61
mplib.collision_detection.fcl
    module, 65
mplib.kinematics.kdl
    module, 98
mplib.kinematics.pinocchio
    module, 91
mplib.planning.ompl
    module, 99
mplib.urdf_utils
    module, 44

N

n (*mplib.collision_detection.fcl.Halfspace property*), 85
n (*mplib.collision_detection.fcl.Plane property*), 87
name (*mplib.ArticulatedModel property*), 39
name (*mplib.collision_detection.AllowedCollision property*), 61
name (*mplib.collision_detection.fcl.FCLModel property*), 82
name (*mplib.collision_detection.fcl.FCLObject property*), 83
name (*mplib.collision_detection.fcl.GJKSolverType property*), 84
nearest_points (*mplib.collision_detection.fcl.DistanceResult property*), 77
NEVER (*mplib.collision_detection.AllowedCollision attribute*), 61
normal (*mplib.collision_detection.fcl.Contact property*), 73
normal (*mplib.collision_detection.fcl.ContactPoint property*), 73
num_contacts() (*mplib.collision_detection.fcl.CollisionResult method*), 71
num_cost_sources() (*mplib.collision_detection.fcl.CollisionResult method*), 71
num_faces (*mplib.collision_detection.fcl.BVHModel property*), 67
num_vertices (*mplib.collision_detection.fcl.BVHModel property*), 67

O

object_name1 (*mplib.collision_detection.WorldCollisionResult property*), 64
object_name1 (*mplib.collision_detection.WorldDistanceResult property*), 65
object_name2 (*mplib.collision_detection.WorldCollisionResult property*), 64
object_name2 (*mplib.collision_detection.WorldDistanceResult property*), 65
OcTree (*class in mplib.collision_detection.fcl*), 85
OMPLPlanner (*class in mplib.planning.ompl*), 100

P

p (*mplib.Pose property*), 43
pad_move_group_qpos() (*mplib.Planner method*), 22
pad_move_group_qpos() (*mplib.sapien_utils.SapienPlanner method*), 57
penetration_depth (*mplib.collision_detection.fcl.Contact property*), 73
penetration_depth (*mplib.collision_detection.fcl.ContactPoint property*), 73
PinocchioModel (*class in mplib.kinematics.pinocchio*), 91
plan() (*mplib.planning.ompl.OMPLPlanner method*), 100
plan_pose() (*mplib.Planner method*), 26
plan_pose() (*mplib.sapien_utils.SapienPlanner method*), 57
plan_qpos() (*mplib.Planner method*), 26
plan_qpos() (*mplib.sapien_utils.SapienPlanner method*), 58
plan_screw() (*mplib.Planner method*), 27
plan_screw() (*mplib.sapien_utils.SapienPlanner method*), 58
Plane (*class in mplib.collision_detection.fcl*), 86
Planner (*class in mplib*), 21
PlanningWorld (*class in mplib*), 28

pos (*mplib.collision_detection.fcl.Contact property*), 73
pos (*mplib.collision_detection.fcl.ContactPoint property*), 73
Pose (*class in mplib*), 42
pose (*mplib.ArticulatedModel property*), 39
pose (*mplib.AttachedBody property*), 41
pose (*mplib.collision_detection.fcl.CollisionObject property*), 70
pose (*mplib.collision_detection.fcl.FCLObject property*), 83
print_attached_body_pose() (*mplib.PlanningWorld method*), 35
print_attached_body_pose() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
print_frames() (*mplib.kinematics.pinocchio.PinocchioModel method*), 98

Q

q (*mplib.Pose property*), 43
qpos (*mplib.ArticulatedModel property*), 39

R

radii (*mplib.collision_detection.fcl.Ellipsoid property*), 78
radius (*mplib.collision_detection.fcl.Capsule property*), 69
radius (*mplib.collision_detection.fcl.Cone property*), 72
radius (*mplib.collision_detection.fcl.Cylinder property*), 76
radius (*mplib.collision_detection.fcl.Sphere property*), 88
remove_articulation() (*mplib.PlanningWorld method*), 35
remove_articulation() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
remove_collision_pairs_from_srdf() (*mplib.collision_detection.fcl.FCLModel method*), 82
remove_default_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 62
remove_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 63
remove_object() (*mplib.Planner method*), 26
remove_object() (*mplib.PlanningWorld method*), 36
remove_object() (*mplib.sapien_utils.SapienPlanner method*), 59
remove_object() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
remove_point_cloud() (*mplib.Planner method*), 24
remove_point_cloud() (*mplib.sapien_utils.SapienPlanner method*), 59
replace_urdf_package_keyword() (*in module mplib.urdf_utils*), 44
res (*mplib.collision_detection.WorldCollisionResult property*), 64
res (*mplib.collision_detection.WorldDistanceResult property*), 65

S

SapienPlanner (*class in mplib.sapien_utils*), 55
SapienPlanningWorld (*class in mplib.sapien_utils*), 45
set() (*mplib.collision_detection.fcl.Triangle method*), 89
set_articulation_planned() (*mplib.PlanningWorld method*), 36
set_articulation_planned() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
set_base_pose() (*mplib.ArticulatedModel method*), 39
set_base_pose() (*mplib.Planner method*), 25
set_base_pose() (*mplib.sapien_utils.SapienPlanner method*), 59
set_default_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 63
set_entry() (*mplib.collision_detection.AllowedCollisionMatrix method*), 63
set_global_seed() (*in module mplib*), 44
set_joint_order() (*mplib.kinematics.pinocchio.PinocchioModel method*), 98
set_link_order() (*mplib.collision_detection.fcl.FCLModel method*), 82
set_link_order() (*mplib.kinematics.pinocchio.PinocchioModel method*), 98
set_move_group() (*mplib.ArticulatedModel method*), 39
set_p() (*mplib.Pose method*), 43
set_pose() (*mplib.ArticulatedModel method*), 40
set_pose() (*mplib.AttachedBody method*), 41
set_pose() (*mplib.collision_detection.fcl.CollisionObject method*), 70
set_q() (*mplib.Pose method*), 43
set_qpos() (*mplib.ArticulatedModel method*), 40
set_qpos() (*mplib.PlanningWorld method*), 36
set_qpos() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
set_qpos_all() (*mplib.PlanningWorld method*), 36
set_qpos_all() (*mplib.sapien_utils.SapienPlanningWorld method*), 54
set_touch_links() (*mplib.AttachedBody method*), 42
shape_poses (*mplib.collision_detection.fcl.FCLObject property*), 83
shapes (*mplib.collision_detection.fcl.FCLObject property*), 83
side (*mplib.collision_detection.fcl.Box property*), 68

signed_distance() (*mplib.collision_detection.fcl.Halfspace method*), 85
signed_distance() (*mplib.collision_detection.fcl.Plane method*), 88
simplify_path() (*mplib.planning.ompl.OMPLPlanner method*), 100
Sphere (*class in mplib.collision_detection.fcl*), 88

T

to_transformation_matrix() (*mplib.Pose method*), 43
TOPP() (*mplib.Planner method*), 24
TOPP() (*mplib.sapien_utils.SapienPlanner method*), 56
total_cost (*mplib.collision_detection.fcl.CostSource property*), 75
tree_IK_NR_JL() (*mplib.kinematics.kdl.KDLModel method*), 99
Triangle (*class in mplib.collision_detection.fcl*), 89
TriangleP (*class in mplib.collision_detection.fcl*), 89

U

update_attached_box() (*mplib.Planner method*), 25
update_attached_box() (*mplib.sapien_utils.SapienPlanner method*), 59
update_attached_mesh() (*mplib.Planner method*), 25
update_attached_mesh() (*mplib.sapien_utils.SapienPlanner method*), 59
update_attached_object() (*mplib.Planner method*), 24
update_attached_object() (*mplib.sapien_utils.SapienPlanner method*), 60
update_attached_sphere() (*mplib.Planner method*), 24
update_attached_sphere() (*mplib.sapien_utils.SapienPlanner method*), 60
update_collision_objects() (*mplib.collision_detection.fcl.FCLModel method*), 82
update_from_simulation() (*mplib.sapien_utils.SapienPlanner method*), 55
update_from_simulation() (*mplib.sapien_utils.SapienPlanningWorld method*), 45
update_point_cloud() (*mplib.Planner method*), 24
update_point_cloud() (*mplib.sapien_utils.SapienPlanner method*), 60
update_pose() (*mplib.AttachedBody method*), 42
update_SRDF() (*mplib.ArticulatedModel method*), 40

V

value (*mplib.collision_detection.AllowedCollision property*), 61
value (*mplib.collision_detection.fcl.GJKSolverType property*), 84
value (*mplib.planning.ompl.FixedJoint property*), 99

W

WorldCollisionResult (*class in mplib.collision_detection*), 64
WorldDistanceResult (*class in mplib.collision_detection*), 64
wrap_joint_limit() (*mplib.Planner method*), 22
wrap_joint_limit() (*mplib.sapien_utils.SapienPlanner method*), 60