robotics_sem

Release 3.1.2025

Ondrej Vita, Josef Kahoun, Simon Pilc

Jan 03, 2025

CONTENTS:

Documentation for semestral work from Robotics course on FEE CTU.

CHAPTER

ONE

ROBOTICS_SEM

1.1 lib package

1.1.1 Subpackages

lib.robotics_toolbox package

Subpackages

lib.robotics_toolbox.core package

Submodules

lib.robotics_toolbox.core.se2 module

Module for representing 2D transformation.

class `lib.robotics_toolbox.core.se2.SE2`(*translation: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *rotation:* SO2 | *float* | *None = None*)

Bases: `object`

Transformation in 2D that is composed of rotation and translation.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Transform given 2D vector by this SE2 transformation.

homogeneous() → ndarray

Return homogeneous transformation matrix.

inverse() → *SE2*

Compute inverse of the transformation. Do not use np.linalg.inv.

set_from(*other:* SE2)

Copy the properties into current instance.

lib.robotics_toolbox.core.se3 module

Module for representing 3D transformation.

class `lib.robotics_toolbox.core.se3.`**SE3**(*translation: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *rotation:* SO3 *| None = None*)

Bases: `object`

Transformation in 2D that is composed of rotation and translation.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Rotate given 3D vector by this transformation.

homogeneous() → ndarray

Return homogeneous matrix representation of the transformation.

inverse() → *SE3*

Compute inverse of the transformation

set_from(*other:* SE3)

Copy the properties into current instance.

lib.robotics_toolbox.core.so2 module

Module for representing 2D rotation.

class `lib.robotics_toolbox.core.so2.`**SO2**(*angle: float = 0.0*)

Bases: `object`

This class represents an SO2 rotations internally represented by rotation matrix.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Rotate given vector by this transformation.

property angle: float

Return angle [rad] from the internal rotation matrix representation.

inverse() → *SO2*

Return inverse of the transformation. Do not change internal property of the object.

lib.robotics_toolbox.core.so3 module

Module for representing 3D rotation.

class `lib.robotics_toolbox.core.so3.`**SO3**(*rotation_matrix: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*)

Bases: `object`

This class represents an SO3 rotations internally represented by rotation matrix.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Rotate given vector by this transformation.

static exp(*rot_vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → *SO3*

Compute SO3 transformation from a given rotation vector, i.e. exponential representation of the rotation.

static from_angle_axis(*angle: float*, *axis: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → *SO3*

Compute rotation from angle axis representation.

static from_euler_angles(*angles: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *seq: list[str]*) → *SO3*

Compute rotation from euler angles defined by a given sequence. angles: is a three-dimensional array of angles seq: is a list of axis around which angles rotate, e.g. 'xyz', 'xzx', etc.

static from_quaternion(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → *SO3*

Compute rotation from quaternion in a form [qx, qy, qz, qw].

inverse() → *SO3*

Return inverse of the transformation.

log() → ndarray

Compute rotation vector from this SO3

static rx(*angle: float*) → *SO3*

Return rotation matrix around x axis.

static ry(*angle: float*) → *SO3*

Return rotation matrix around y axis.

static rz(*angle: float*) → *SO3*

Return rotation matrix around z axis.

to_angle_axis() → tuple[float, ndarray]

Compute angle axis representation from self.

to_quaternion() → ndarray

Compute quaternion from self.

Module contents

Core functionality for the robotics_toolbox package is defined by transformations in 2D and 3D space. This module implements these transformations as classes that can be used to represent and manipulate rotations and translations in 2D and 3D space.

lib.robotics_toolbox.planning package

Submodules

lib.robotics_toolbox.planning.prm module

Module for path planning using Probabilistic Roadmap Method (PRM).

class lib.robotics_toolbox.planning.prm.**GraphPlanner**(*graph_matrix: list*)

Bases: object

get_path(*i*, *j*) → list

will get path from node i to j as list of nodes IDs visited

class lib.robotics_toolbox.planning.prm.**Node**(*id: int*, *config: ArrayLike* | SE2 | SE3)

Bases: object

add_neighbour(*neighbour:* Node, *path_to_neighbour: list[ArrayLike* | SE2 | SE3*]*)

class lib.robotics_toolbox.planning.prm.**PRM**(*robot:* RobotBase, *delta_q=0.2*)

Bases: object

closest_connect(*q: ArrayLike* | SE2 | SE3, *q_to_graph: bool = True*)

connect the closest node to the given configuration

return: path from the given config to the closest reachable node and closest reacheble node id

connect(*q_init: ArrayLike* | SE2 | SE3, *q_goal: ArrayLike* | SE2 | SE3, *max_iter: int = 1000*) → list[ArrayLike | *SE2* | *SE3*] | None

will find path between two given configurations

explore(*max_nodes: int = 500*) → None

PRM algorithm for motion planning.

plan(*q_start: ArrayLike | SE2 | SE3*, *q_goal: ArrayLike | SE2 | SE3*, *graph_planner: GraphPlanner = <class 'lib.robotics_toolbox.planning.prm.GraphPlanner'>*) → list[ArrayLike | *SE2* | *SE3*]

will plan path in the current graph

lib.robotics_toolbox.planning.rrt module

Module for RRT motion planning algorithm.

class lib.robotics_toolbox.planning.rrt.**RRT**(*robot:* RobotBase, *delta_q=0.2*, *p_sample_goal=0.5*)

Bases: object

plan(*q_start: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]* | SE2 | SE3, *q_goal: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]* | SE2 | SE3, *max_iterations: int = 10000*) → list[_SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | *SE2* | *SE3*]

RRT algorithm for motion planning.

random_shortcut(*path: list[ndarray* | SE2 | SE3*]*, *max_iterations=100*) → list[ndarray | *SE2* | *SE3*]

Random shortcut algorithm that pick two points on the path randomly and tries to interpolate between them. If collision free interpolation exists, the path between selected points is replaced by the interpolation.

class lib.robotics_toolbox.planning.rrt.**TreeList**

Bases: object

add_node(*node:* TreeNode) → None

calculate_distance(*q1: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] |* SE2 | SE3, *q2: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] |* SE2 | SE3) → float

return_nearest_node(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] |* SE2 | SE3) → *TreeNode*

class lib.robotics_toolbox.planning.rrt.**TreeNode**(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] |* SE2 | SE3, *parent:* TreeNode *| None = None*)

Bases: object

return_parent()

return_path_from_start()

Module contents

Module for path planning algorithms.

lib.robotics_toolbox.render package

Submodules

lib.robotics_toolbox.render.mobile_robot_renderer module

class lib.robotics_toolbox.render.mobile_robot_renderer.**MobileRobotRenderer**(*ax: Axes*, *robot:* MobileRobot)

Bases: object

update()

lib.robotics_toolbox.render.planar_manipulator_renderer module

class lib.robotics_toolbox.render.planar_manipulator_renderer.**PlanarManipulatorRenderer**(*ax: Axes*, *robot:* Pla-n-ar-Ma-nip-u-la-tor, ***kwargs*)

Bases: object

plot_init(*color='k'*, *lw=3*, *ms=7*)

Plot the robot. It returns the plt_objects, that you can use to update the robot pose in animation.

plot_line_between_points(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, **args*, ***kwargs*)

Plot line between two given 2D points a and b. Other arguments passed to ax.plot function.

update()

lib.robotics_toolbox.render.renderer_planar module

class lib.robotics_toolbox.render.renderer_planar.**RendererPlanar**(*xlim: tuple[float, float] = (-1, 1)*, *ylim: tuple[float, float] = (-1, 1)*, *lim_scale: float = 1.0*)

Bases: `object`

plot_line_between_points(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, **args*, ***kwargs*)

Plot line between two given 2D points a and b. Other arguments passed to ax.plot function.

plot_manipulator(*robot: PlanarManipulator*, ***kwargs*)

plot_mobile_robot(*robot: MobileRobot*)

Plot mobile robot into the figure. If this function is called multiple times for the same robot, this function redraw the robot to the new pose instead of drawing a new one.

plot_se2(*t: SE2*, *length=0.1*, **args*, ***kwargs*)

Plot SE2 frame.

plot_so2(*t: SO2*, *length=0.1*, **args*, ***kwargs*)

Plot SO2 frame in the origin.

redraw_all()

Redraw all the manipulators that has been plotted before.

static **wait_for_close**()

static **wait_for_enter**(*msg: str | None = None*)

lib.robotics_toolbox.render.renderer_spatial module

class lib.robotics_toolbox.render.renderer_spatial.**RendererSpatial**

Bases: `Scene`

plot_drone(*robot: Drone*, *render=True*)

plot_manipulator(*robot: SpatialManipulator*, *render=True*)

plot_se3(*t: SE3*, *scale=1.0*, *render=True*)

wait_at_the_end()

A method that just sleep for a few seconds. Call it at the end to prevent interruption of the connection with the renderer.

static **wait_for_enter**(*msg: str | None = None*)

lib.robotics_toolbox.render.se2_renderer module

class lib.robotics_toolbox.render.se2_renderer.**SE2Renderer**(*ax: Axes*, *t: SE2 | SO2*, *length: float = 0.1*, **args*, ***kwargs*)

Bases: `object`

plot_line_between_points(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, **args*, ***kwargs*)

Plot line between two given 2D points a and b. Other arguments passed to ax.plot function.

update()

Module contents

Module for rendering robot models in 2D (matplotlib) and 3D (robomeshcat).

class lib.robotics_toolbox.render.**RendererPlanar**(*xlim: tuple[float, float] = (-1, 1)*, *ylim: tuple[float, float] = (-1, 1)*, *lim_scale: float = 1.0*)

Bases: `object`

plot_line_between_points(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, **args*, ***kwargs*)

Plot line between two given 2D points a and b. Other arguments passed to ax.plot function.

plot_manipulator(*robot: PlanarManipulator*, ***kwargs*)

plot_mobile_robot(*robot: MobileRobot*)

Plot mobile robot into the figure. If this function is called multiple times for the same robot, this function redraw the robot to the new pose instead of drawing a new one.

plot_se2(*t: SE2*, *length=0.1*, **args*, ***kwargs*)

Plot SE2 frame.

plot_so2(*t: SO2*, *length=0.1*, **args*, ***kwargs*)

Plot SO2 frame in the origin.

redraw_all()

Redraw all the manipulators that has been plotted before.

static **wait_for_close**()

static **wait_for_enter**(*msg: str | None = None*)

class lib.robotics_toolbox.render.**RendererSpatial**

Bases: Scene

plot_drone(*robot:* Drone, *render=True*)

plot_manipulator(*robot:* SpatialManipulator, *render=True*)

plot_se3(*t:* SE3, *scale=1.0*, *render=True*)

wait_at_the_end()

A method that just sleep for a few seconds. Call it at the end to prevent interruption of the connection with the renderer.

static **wait_for_enter**(*msg: str | None = None*)

lib.robotics_toolbox.robots package

Submodules

lib.robotics_toolbox.robots.drone module

class lib.robotics_toolbox.robots.drone.**Drone**

Bases: RobotBase

configuration() → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

in_collision() → bool

Check if robot is in collision.

sample_configuration() → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

set_configuration(*configuration: ndarray* | SE2 | SE3)

Set internal configuration to @param configuration. Returns self.

lib.robotics_toolbox.robots.mobile_robot module

class lib.robotics_toolbox.robots.mobile_robot.**MobileRobot**(*size: float = 0.3*)

Bases: RobotBase

configuration() → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

in_collision() → bool

Check if robot is in collision.

sample_configuration() → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

set_configuration(*configuration: ndarray* | SE2 | SE3)

Set internal configuration to @param configuration. Returns self.

lib.robotics_toolbox.robots.planar_manipulator module

Module for representing planar manipulator.

class `lib.robotics_toolbox.robots.planar_manipulator.`**`PlanarManipulator`**(*link_parameters: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *structure: list[str] | str | None = None*, *base_pose:* SE2 | *None = None*, *gripper_length: float = 0.2*)

Bases: `RobotBase`

`configuration`() → ndarray | *SE2* | *SE3*

Get the robot configuration.

`property dof`

Return number of degrees of freedom.

`fk_all_links`() → list[*SE2*]

Compute FK for frames that are attached to the links of the robot. The first frame is base_frame, the next frames are described in the constructor.

`flange_pose`() → *SE2*

Return the pose of the flange in the reference frame.

`ik_analytical`(*flange_pose_desired:* SE2) → list[ndarray]

Compute IK analytically, return all solutions for joint limits being from -pi to pi for revolute joints -inf to inf for prismatic joints.

`ik_numerical`(*flange_pose_desired:* SE2, *max_iterations=1000*, *acceptable_err=0.0001*) → bool

Compute IK numerically. Value self.q is used as an initial guess and updated to solution of IK. Returns True if converged, False otherwise.

`in_collision`() → bool

Check if robot in its current pose is in collision.

`jacobian`() → ndarray

Computes jacobian of the manipulator for the given structure and configuration.

`jacobian_finite_difference`(*delta=1e-05*) → ndarray

`sample_configuration`()

Sample robot configuration inside the configuration space. Will change internal state.

`set_configuration`(*configuration: ndarray* | SE2 | SE3)

Set configuration of the robot, return self for chaining.

lib.robotics_toolbox.robots.planar_manipulator_dynamics module

class `lib.robotics_toolbox.robots.planar_manipulator_dynamics.`**`PlanarManipulatorDynamics`**(*masses=None*, **args*, ***kwargs*)

Bases: `PlanarManipulator`

`constrained_forward_dynamics`(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *dq: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *tau: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *damping: float = 0.0*) → ndarray

Implement constrained forward dynamics of the robot. I.e. compute ddq from q, dq, tau and damping. Use eq. of motion: tau = M(q) ddq + h(q,dq) + damping * dq + A^T lambda. The constraint is fixed, such that end effector moves along line with angle 45deg w.r.t. x-axis of reference frame.

`forward_dynamics`(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *dq: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *tau: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *damping: float = 0.0*) → ndarray

Implement forward dynamics of the robot. I.e. compute ddq from q, dq, tau and damping. Use eq. of motion: tau = M(q) ddq + h(q,dq) + damping * dq.

`h`(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *dq: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Coriolis and gravity terms at configuration q and velocity dq.

`inverse_dynamics`(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *dq: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *ddq: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *damping: float = 0.0*) → ndarray

Implement inverse dynamics of the robot. I.e. compute tai from q, dq, ddq, and damping. Use eq. of motion: tau = M(q) ddq + h(q,dq) + damping * dq.

`mass_matrix`(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Mass matrix of the robot at configuration q.

lib.robotics_toolbox.robots.robot_base module

class `lib.robotics_toolbox.robots.robot_base.`**`RobotBase`**

Bases: `object`

abstract configuration() → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

abstract in_collision() → bool

Check if robot is in collision.

abstract sample_configuration() → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

abstract set_configuration(*configuration: ndarray* | SE2 | SE3)

Set internal configuration to @param configuration. Returns self.

lib.robotics_toolbox.robots.spatial_manipulator module

class `lib.robotics_toolbox.robots.spatial_manipulator.`**`SpatialManipulator`**(*robot_name: str* | *None = None*, *urdf_path: str* | *Path* | *None = None*, *mesh_folder_path: Path* | *str* | *list[Path]* | *list[str]* | *None = None*, *srdf_path: str* | *Path* | *None = None*, *base_pose:* SE3 | *None = None*, ***kwargs*)

Bases: `RobotBase`

configuration() → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

property dof: int

Return number of degrees of freedom for the robot.

flange_pose(*flange_link_name: str* | *None = None*) → *SE3*

Return a flange pose defined by the link name. Flange link name can be empty for Panda robot.

in_collision() → bool

Check if robot is in collision.

jacobian(*flange_link_name: str* | *None = None*) → ndarray

Computes jacobian of the manipulator for the given structure and configuration.

sample_configuration() → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

set_configuration(*configuration: ndarray* | SE2 | SE3)

Set internal configuration to @param configuration. Returns self.

Module contents

This module contains the robot classes for mobile robots, manipulators, and drones.

class `lib.robotics_toolbox.robots.`**`Drone`**

Bases: `RobotBase`

`configuration()` → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

`in_collision()` → bool

Check if robot is in collision.

`sample_configuration()` → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

`set_configuration`(*configuration: ndarray* | SE2 | SE3)

Set internal configuration to @param configuration. Returns self.

class `lib.robotics_toolbox.robots.`**`MobileRobot`**(*size: float = 0.3*)

Bases: `RobotBase`

`configuration()` → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

`in_collision()` → bool

Check if robot is in collision.

`sample_configuration()` → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

`set_configuration`(*configuration: ndarray* | SE2 | SE3)

Set internal configuration to @param configuration. Returns self.

class `lib.robotics_toolbox.robots.`**`PlanarManipulator`**(*link_parameters: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None, structure: list[str] | str | None = None, base_pose:* SE2 *| None = None, gripper_length: float = 0.2*)

Bases: `RobotBase`

`configuration()` → ndarray | *SE2* | *SE3*

Get the robot configuration.

property dof

Return number of degrees of freedom.

`fk_all_links()` → list[*SE2*]

Compute FK for frames that are attached to the links of the robot. The first frame is base_frame, the next frames are described in the constructor.

`flange_pose()` → *SE2*

Return the pose of the flange in the reference frame.

ik_analytical(*flange_pose_desired: SE2*) → list[ndarray]

Compute IK analytically, return all solutions for joint limits being from -pi to pi for revolute joints -inf to inf for prismatic joints.

ik_numerical(*flange_pose_desired: SE2*, *max_iterations=1000*, *acceptable_err=0.0001*) → bool

Compute IK numerically. Value self.q is used as an initial guess and updated to solution of IK. Returns True if converged, False otherwise.

in_collision() → bool

Check if robot in its current pose is in collision.

jacobian() → ndarray

Computes jacobian of the manipulator for the given structure and configuration.

jacobian_finite_difference(*delta=1e-05*) → ndarray

sample_configuration()

Sample robot configuration inside the configuration space. Will change internal state.

set_configuration(*configuration: ndarray | SE2 | SE3*)

Set configuration of the robot, return self for chaining.

class lib.robotics_toolbox.robots.**SpatialManipulator**(*robot_name: str | None = None*, *urdf_path: str | Path | None = None*, *mesh_folder_path: Path | str | list[Path] | list[str] | None = None*, *srdf_path: str | Path | None = None*, *base_pose: SE3 | None = None*, ***kwargs*)

Bases: RobotBase

configuration() → ndarray | *SE2* | *SE3*

Get the configuration of the robot, can be array, SE2, or SE3.

property dof: int

Return number of degrees of freedom for the robot.

flange_pose(*flange_link_name: str | None = None*) → *SE3*

Return a flange pose defined by the link name. Flange link name can be empty for Panda robot.

in_collision() → bool

Check if robot is in collision.

jacobian(*flange_link_name: str | None = None*) → ndarray

Computes jacobian of the manipulator for the given structure and configuration.

sample_configuration() → ndarray | *SE2* | *SE3*

Sample robot configuration inside the configuration space.

set_configuration(*configuration: ndarray | SE2 | SE3*)

Set internal configuration to @param configuration. Returns self.

lib.robotics_toolbox.utils package

Submodules

lib.robotics_toolbox.utils.animation_utils module

lib.robotics_toolbox.utils.animation_utils.**create_gif_from_mp4**(*input_vid: Path | str*, *output: str | Path | None = None*)

Convert input_vid (mp4) to GIF by first generating the color pallet.

lib.robotics_toolbox.utils.animation_utils.**create_mp4_from_folder**(*folder: Path | str = '/tmp/animation'*, *output: str | Path | None = None*, *fps: int = 10*)

From the folder that contains images names img_X.png, create mp4 animation.

lib.robotics_toolbox.utils.animation_utils.**save_fig**(*output_folder: Path | str = '/tmp/animation'*, *renderer=None*)

Save fig of the renderer into the given output folder. Output folder is cleaned on the first run of this command. If renderer not provided use plt.savefig. This name of figures is img_{id}.png.

lib.robotics_toolbox.utils.configuration_utils module

lib.robotics_toolbox.utils.configuration_utils.**distance_between_configurations**(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]* | SE2 | SE3, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]* | SE2 | SE3) → float

Compute distance between two configurations, expressed either in task-space SE2/SE3 or joint space np.ndarray `lib.robotics_toolbox.utils.configuration_utils.`**`interpolate`**`(`*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]* | SE2 | SE3, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]* | SE2 | SE3, *d: float*`)` → ndarray | *SE2* | *SE3*

Interpolate between two configurations, s.t. dist(a,b) = d

lib.robotics_toolbox.utils.geometry_utils module

`lib.robotics_toolbox.utils.geometry_utils.`**`circle_circle_intersection`**`(`*c0: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes], r0: float, c1: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes], r1: float*`)` → list[ndarray]

Computes intersection of the circles defined by center c_i and radius r_i. Returns empty array if there is no solution, two solutions otherwise. If there are infinite number of solutions, select two (almost) randomly.

`lib.robotics_toolbox.utils.geometry_utils.`**`circle_line_intersection`**(*c: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *r: float*, *a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → list[ndarray]

Compute intersection of circle (c, r) with line defined by two points (a, b)

`lib.robotics_toolbox.utils.geometry_utils.`**`nullspace`**(*A*, *atol=1e-13*, *rtol=0.0*)

Compute kernel, i.e. nullspace of the given matrix A.

Module contents

Module with utility functions for the robotics_toolbox package.

`lib.robotics_toolbox.utils.`**`circle_circle_intersection`**(*c0: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *r0: float*, *c1: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *r1: float*) → list[ndarray]

Computes intersection of the circles defined by center c_i and radius r_i. Returns empty array if there is no solution, two solutions otherwise. If there are infinite number of solutions, select two (almost) randomly.

lib.robotics_toolbox.utils.**circle_line_intersection**(*c: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *r: float*, *a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → list[ndarray]

Compute intersection of circle (c, r) with line defined by two points (a, b)

lib.robotics_toolbox.utils.**create_gif_from_mp4**(*input_vid: Path | str*, *output: str | Path | None = None*)

Convert input_vid (mp4) to GIF by first generating the color pallet.

lib.robotics_toolbox.utils.**create_mp4_from_folder**(*folder: Path | str = '/tmp/animation'*, *output: str | Path | None = None*, *fps: int = 10*)

From the folder that contains images names img_X.png, create mp4 animation.

lib.robotics_toolbox.utils.**distance_between_configurations**(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | SE2 | SE3*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | SE2 | SE3*) → float

Compute distance between two configurations, expressed either in task-space SE2/SE3 or joint space np.ndarray

lib.robotics_toolbox.utils.**interpolate**(*a: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | SE2 | SE3*, *b: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | SE2 | SE3*, *d: float*) → ndarray | *SE2* | *SE3*

Interpolate between two configurations, s.t. dist(a,b) = d

lib.robotics_toolbox.utils.**nullspace**(*A*, *atol=1e-13*, *rtol=0.0*)

Compute kernel, i.e. nullspace of the given matrix A.

lib.robotics_toolbox.utils.**save_fig**(*output_folder: Path | str = '/tmp/animation'*, *renderer=None*)

Save fig of the renderer into the given output folder. Output folder is cleaned on the first run of this command. If renderer not provided use plt.savefig. This name of figures is img_{id}.png.

Module contents

Robotics Toolbox is a Python library for robot kinematics, dynamics, and control. However, it is not complete, some of the implementations are missing and your goal is to complete them. To verify the correctness of your implementation, you can run the tests in the tests folder.

1.1.2 Submodules

1.1.3 lib.base module

Module for storing infromation about the base of the cubes.

class lib.base.**Base**

Bases: `object`

Class for storing information about the base of the cubes.

- aruco_ids

List[int] - List of ArUco IDs

- cube_centers

List[List[float]] - List of cube centers (list of [x, y])

- position

SE3 - Tranformation matrix from base to the camera (T_CB)

- target_transforms

List[SE3] - List of target transformation matrices from the cube positions to the camera (T_CT)

csv2base(*filename: str*) → None

Loads base data from a CSV file.

Parameters

filename (*-*) – str - path to the CSV file describing the base

get_target_transforms_from_camera() → List[*SE3*]

Return the list of target transformation matrices from the cube positions to camera (T_CT).

Returns

- List[SE3] - List of target transformation matrices from cube positions to camera

get_target_transforms_from_robot(*T_RC: SE3*) → List[*SE3*]

Return the list of target transformation matrices from cube positions to robot (T_RT).

Parameters

T_RC (*-*) – SE3 - Transformation matrix from camera to robot (T_RC)

Returns

- List[SE3] - List of target transformation matrices from cube positions to robot

set_position(*position: SE3*) → None

Sets the position of the base.

Parameters

position (*-*) – SE3 - Tranformation matrix from base to the camera (T_CB) (calculated using base_pose_estimator)

1.1.4 lib.base_pose_estimator module

Module for estimating the pose of the base of the cubes in relation to the camera.

class `lib.base_pose_estimator.`**`BasePoseEstimator`**(*camera_matrix: ndarray | None = None*, *distortion_coefficients: ndarray | None = None*)

Bases: `object`

This class serves for estimating the pose of the base of the cubes in relation to the camera.

- camera_matrix
np.ndarray - camera matrix

- distortion_coefficients
np.ndarray - distortion coefficients

- aruco_size
float - size of the aruco markers

- base
Base - base object for which the pose is estimated

`calculate_pose`(*camera*) → bool

`calculate_pose_multiple_photos`(*images: list*) → bool

Calculates the pose of the base in relation to the camera using perspective-n-point algorithm.

Parameters
img (-) – list - list of images from the camera

Returns

- bool - True if the pose was calculated, False if the algorithm failed

`calculate_pose_normal`(*img: ndarray*) → bool

Calculates the pose of the base in relation to the camera using perspective-n-point algorithm.

Parameters
img (-) – np.ndarray - image from the camera

Returns

- bool - True if the pose was calculated, False if the algorithm failed

`get_updated_base`() → *Base*

Returns the base object with the updated pose. Can be called after calculate_pose.

Returns

- Base - base object with the updated pose.

`set_base`(*base:* Base) → None

Sets the base object.

Parameters
base (-) – Base - base object

`set_camera_matrix`(*camera_matrix: ndarray*) → None

Sets the camera matrix.

Parameters
camera_matrix (-) – np.ndarray - camera matrix

set_distortion_coefficients(*distortion_coefficients: ndarray*) → None

Sets the distortion coefficients.

Parameters
distortion_coefficients (*-*) – np.ndarray - distortion coefficients

set_rotation(*base*, *tvec*, *rvec*)

1.1.5 lib.basler_camera module

1.1.6 lib.camera_calibration module

class lib.camera_calibration.**CameraCalib**

Bases: `object`

class for calibrating the camera using a chessboard or a Charuco board.

calibrate(*json_filename: str*) → None

Calibrate the camera. Save the calibration data to a json file.

Parameters
json_filename (*str*) – Name of the json file to save the calibration data.

calibrate_using_charuco(*json_filename: str*) → None

Calibrate the camera using a Charuco board. Save the calibration data to a json file.

Parameters
json_filename (*str*) – Name of the json file to save the calibration data.

get_images() → list

set_image_path(*img_path: str*) → None

lib.camera_calibration.**blur_laplacian**(*image*, *threshold=250*)

Check if an image is blurry using Laplacian variance.

lib.camera_calibration.**edge_sharpness**(*image: ndarray*, *gradient_threshold=300*) → tuple[bool, float]

Analyze edge sharpness based on gradient size, and disqualify blurry images. :param image: gradient_threshold (float, threshold for gradient sharpness) :type image: np.ndarray :param gradient_threshold: _description_. Defaults to GRADIENT_THRESHOLD. :type gradient_threshold: _type_, optional

Returns
avg_gradient (float, average gradient)

Return type
tuple[bool, float]

1.1.7 lib.display module

class lib.display.**Display**

Bases: `object`

Class to display the positions of the robot in 3D space.

add_positions(*positions: list*)

Add positions to the list of positions.

Parameters
positions (*list*) – list of positions

display_positions()

Displace the positions in 3D space.

setup_positions(*positions: list*)

Positions that will be displayed.

Parameters
positions (*list*) – list of positions

1.1.8 lib.hand_eye_calib module

Module for hand-eye calibration using images of a calibration target taken from the camera.

class lib.hand_eye_calib.**HandEyeCalib**

Bases: object

load_fk_transformation(*hand_eye_calib_data: str*) → List[*SE3*]

Loads the forward kinematics data from a json file.

Parameters
hand_eye_calib_data (*str*) – Path to the hand-eye calibration data

Returns
List of SE3 objects representing the forward kinematics data

Return type
List[*SE3*]

load_fk_transformation_and_points(*hand_eye_calib_data: str*) → Tuple[List[*SE3*], List[array], List[array]]

Loads the forward kinematics data from a json file, and the corners of aruco markers.

Parameters
hand_eye_calib_data (*str*) – Path to the hand-eye calibration data

Returns
List of SE3 objects representing the forward kinematics data, 3D points in target coordinate system and 2D points in camera image

Return type
Tuple[List[*SE3*], List[np.array], List[np.array]]

load_initial_parameters(*hand_eye_calib_results: str*) → array

Loads the initial parameters from a json file.

Parameters
hand_eye_calib_results (*str*) – Path to the initial hand-eye calibration results

Returns
Initial parameters [tvec_RC, rvec_RC, tvec_GT, rvec_GT]

Return type
np.array

prepare_points(*calib_imgs_path: str*) → Tuple[List[array], List[array], List[*SE3*]]

Prepare the 3D and 2D points for the optimization.

Parameters
calib_imgs_path (*-*) – str - Path to the calibration images

Returns

- Tuple[List[np.array], List[np.array]] - 3D points in target coordinate system and 2D points in camera image

`reprojection_error`(*params: array*, *K: array*, *dist_coef: array*, *points_3d: array*, *points_2d: array*, *T_RG: List[SE3]*) → float

Calculates the reprojection error.

Used as the optimazation criterion to calculate the robot to camera (and gripper to target) calibration.

Args:aruco_ids:

- params: np.array - Parameters to optimize (rvecs and tvecs of the transformations) [tvec_RC, rvec_RC, tvec_GT, rvec_GT] (tvecs in meters)
- K: np.array - Camera matrix
- dist_coef: np.array - Distortion coefficients
- points_3d: np.array - 3D points of the calibration target (in the target coordinate system)
- points_2d: np.array - 2D points of the calibration target (the image points)
- T_RG: List[SE3] - List of gripper to robot transformation matrices (from forward kinematics) (in meters)

Returns

- float - Reprojection error

`reprojection_error_ls`(*params: array*, *K: array*, *dist_coef: array*, *points_3d: array*, *points_2d: array*, *T_RG: List[SE3]*) → array

New and revised reprojection error function for least squares optimization.

Used as the optimazation criterion to calculate the robot to camera (and gripper to target) calibration.

Parameters

- **params** (-) – np.array - Parameters to optimize (rvecs and tvecs of the transformations) [tvec_RC, rvec_RC, tvec_GT, rvec_GT] (tvecs in meters)
- **K** (-) – np.array - Camera matrix
- **dist_coef** (-) – np.array - Distortion coefficients
- **points_3d** (-) – np.array - 3D points of the calibration target (in the target coordinate system)
- **points_2d** (-) – np.array - 2D points of the calibration target (the image points)
- **T_RG** (-) – List[SE3] - List of gripper to robot transformation matrices (from forward kinematics) (in meters)

Returns

- np.array - Reprojection error residuals

`reprojection_optimization`(*camera_calib_filename: str*, *hand_eye_calib_data: str*, *hand_eye_calib_results: str*) → bool

Optimizes the camera to robot and gripper to target calibration using reprojection error.

Parameters

- **camera_calib_filename** – (str) Camera calibration filename
- **hand_eye_calib_data** – (str) Path to the hand-eye calibration data

- **hand_eye_calib_results** – (str) Path to the initial hand-eye calibration results

Returns
True if the optimization was successful, False otherwise

Return type
bool

1.1.9 lib.robot_wrapper module

class lib.robot_wrapper.**Robot**(*robotHome: bool = True*)

Bases: CRS97

get_desired_pos_of_custom_gripper(*pos:* SE3) → *SE3*

This function returns the desired position of the robot in the robot base coordinates,
for the custom gripper.

Parameters
pos (SE3) – position of the robot in the camera coordinates.

Returns
Position of the custom gripper in the robot base coordinates, or None if the position is incorrect.

Return type
SE3

get_flange_pos() → ndarray

This function returns the position of the flange (not the gripper) in the robot base coordinates.

Returns
(x, y, z) coordinates of the flange.

Return type
np.ndarray

get_gripper_pos() → ndarray

This function returns the T transformation matrix of the gripper.

Returns
T matrix of the flange.

Return type
np.ndarray

get_name() → str

returns name of the robot.

Returns
name of the robot

Return type
str

gripper_drop()

Closes the gripper using the learned strength.

gripper_pick()

Opens the gripper.

move_custom_gripper(*pos: SE3*)

This function moves the robot to the given position in robot coordinates.
moves the custom gripper to a given position.

Parameters
pos (SE3) – position to which the custom gripper will be moved

Returns
True if the robot moved to the position, False otherwise.

Return type
bool

move_to_angles(*q: ndarray*) → bool

moves the robot to the given position based on a given joint angles.

Parameters
q (*np.ndarray*) – joint angles of the robot

Returns
True if the robot moved to the position, False otherwise.

Return type
bool

move_to_camera_capture_pos() → bool

This function moves the robot to a position in which shadows are minimized in the camera view.

Returns
True if the robot moved to the gate position, False otherwise.

Return type
bool

move_to_coordinates(*pos: ndarray*) → bool

This function moves the robot to the given position.

Parameters
pos (*np.ndarray*) – Transformation matrix of the gripper. T matrix.

Returns
True if the robot moved to the position, False otherwise.

Return type
bool

move_to_gate() → bool

This function moves the robot to the gate position, which makes easier access to the gripper for making changes.

Returns
True if the robot moved to the gate position, False otherwise.

Return type
bool

move_under_camera() → bool

Moves the robot under a camera.

Returns
True if the robot moved under the camera, False otherwise.

Return type
bool

pick_drop_part(*base: Base*, *index_of_target: int*, *pick_part: bool*)

This function picks or drops a part from a given base.

Parameters

- **base** (Base) – Base object from which the part will be picked or dropped.
- **index_of_target** (*int*) – Index of the target part.
- **pick_part** (*bool*) – True if the part will be picked, False if the part will be dropped.

pick_drop_part_with_wiggle(*base: Base*, *index_of_target: int*, *pick_part: bool*)

This function picks or drops a part from a given base and wiggles to make sure the part falls into the holder.

Parameters

- **base** (Base) – Base object from which the part will be picked or dropped.
- **index_of_target** (*int*) – Index of the target part.
- **pick_part** (*bool*) – True if the part will be picked, False if the part will be dropped.

position_with_adaptive_offset(*pos: ndarray*) → ndarray

Transforms the position using adaptive offset

Parameters
pos (*np.ndarray*) – Position to be transformed

Returns
Transformed position

Return type
np.ndarray

return_best_ik(*pos: ndarray*) → ndarray

This function returns the best ik solution for the robot to reach the given position.

Parameters
pos (*np.ndarray*) – Transformation matrix of the gripper.

Returns
Best ik solution for the robot.

Return type
np.ndarray

setup(*gripper_offset_matrix: SE3 | None = None*, *cam_to_rob_matrix: SE3 | None = None*)

sets up transformation matrix from camera to robot base. also sets up the offset matrix

Parameters

- **cam_to_rob_matrix** (SE3) – transformation matrix from camera to robot base.
- **grippper_offset_matrix** (SE3 *| None, optional*) – offset matrix to custom gripper. Defaults to None.

wiggle_move_to_coords(*pos: ndarray*) → bool

This function moves the robot to the given position while wiggling.

Parameters
pos (*np.ndarray*) – Transformation matrix of the gripper. T matrix.

Returns
True if the robot moved to the position, False otherwise.

Return type
bool

wiggle_wiggle_wiggle()

This function moves the robot in a wiggle pattern.

Serves as a failsafe when the coordinates of the part are not precise enough. This wiggle position should make the part fall into the holder.

1.1.10 lib.se3 module

Module for representing 3D transformation.

class lib.se3.**SE3**(*translation: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*, *rotation:* SO3 *| None = None*)

Bases: `object`

Transformation in 2D that is composed of rotation and translation.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Rotate given 3D vector by this transformation.

classmethod from_dict(*data*)

homogeneous() → ndarray

Return homogeneous matrix representation of the transformation.

inverse() → *SE3*

Compute inverse of the transformation

set_from(*other:* SE3)

Copy the properties into current instance.

to_dict()

1.1.11 lib.so2 module

Module for representing 2D rotation.

class lib.so2.**SO2**(*angle: float = 0.0*)

Bases: `object`

This class represents an SO2 rotations internally represented by rotation matrix.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Rotate given vector by this transformation.

property angle: float

Return angle [rad] from the internal rotation matrix representation.

inverse() → *SO2*

Return inverse of the transformation. Do not change internal property of the object.

1.1.12 lib.so3 module

Module for representing 3D rotation.

class `lib.so3.SO3`(*rotation_matrix: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None = None*)

Bases: `object`

This class represents an SO3 rotations internally represented by rotation matrix.

act(*vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → ndarray

Rotate given vector by this transformation.

static exp(*rot_vector: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → *SO3*

Compute SO3 transformation from a given rotation vector, i.e. exponential representation of the rotation.

static from_angle_axis(*angle: float, axis: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → *SO3*

Compute rotation from angle axis representation.

static from_euler_angles(*angles: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes], seq: list[str]*) → *SO3*

Compute rotation from euler angles defined by a given sequence. angles: is a three-dimensional array of angles seq: is a list of axis around which angles rotate, e.g. 'xyz', 'xzx', etc.

static from_quaternion(*q: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → *SO3*

Compute rotation from quaternion in a form [qx, qy, qz, qw].

inverse() → *SO3*

Return inverse of the transformation.

log() → ndarray

Compute rotation vector from this SO3

static rx(*angle: float*) → *SO3*

Return rotation matrix around x axis.

static ry(*angle: float*) → *SO3*

Return rotation matrix around y axis.

static rz(*angle: float*) → *SO3*

Return rotation matrix around z axis.

to_angle_axis() → tuple[float, ndarray]

Compute angle axis representation from self.

to_quaternion() → ndarray

Compute quaternion from self.

1.1.13 lib.utils module

`lib.utils.get_R`(*roll*, *pitch*, *yaw*) → ndarray

Returns the rotation matrix from roll, pitch, yaw angles

Parameters

- **roll** – float - roll angle
- **pitch** – float - pitch angle
- **yaw** – float - yaw angle

Returns
rotation matrix

Return type
np.ndarray

`lib.utils.get_images_from_dir`(*img_path: str*) → list

Loads all images from a directory.

Parameters
img_path (*str*) – The path to the directory containing the images.

Returns
A list of paths to the images.

Return type
list

`lib.utils.load_camera_calibration_data`(*filename: str*) → Tuple[ndarray, ndarray, ndarray]

Loads camera calibration data from a file

Parameters
filename – str - filename of the calibration data

Returns
camera matrix, distortion coefficients, new camera matrix

Return type
tuple[np.ndarray, np.ndarray, np.ndarray]

`lib.utils.read_data_from_json`(*file_path*)

Reads multiple lists or dictionaries from a JSON file.

Parameters
file_path (*str*) – The path to the JSON file.

Returns
A dictionary containing the data.

Return type
dict

`lib.utils.robot_gripper_to_SE3`(*robot*) → *SE3*

Gets transformation from gripper to robot using robot joint angles

Parameters
robot – Robot - robot object

Returns
SE3 object representing transformation from gripper to robot

Return type
SE3

`lib.utils.vector_to_se3(tvec, rvec) → SE3`

Converts translation and rotation (Euler) vectors to SE3 transformation class

Parameters

- **tvec** – ArrayLike - translation vector
- **rvec** – ArrayLike - rotation vector

Returns
SE3 object

Return type
SE3

`lib.utils.write_data_to_json(file_path, **data)`

Writes multiple lists or dictionaries to a JSON file.

Parameters

- **file_path** (*str*) – The path to the JSON file.
- ****data** – Arbitrary keyword arguments representing the data to be saved.

1.1.14 Module contents

Library containing all the modules

1.2 src package

1.2.1 Subpackages

src.calibration package

Submodules

src.calibration.camera_calib_new module

Gets the camera image and detects the Aruco markers on it.

`src.calibration.camera_calib_new.main()`

src.calibration.camera_charuco_calibration_routine module

`src.calibration.camera_charuco_calibration_routine.camera_take_image(camera: BaslerCamera, dirname: str)`

Takes images from the camera and saves them in the directory given, until the user presses 'x' to exit. Numbers the images sequentially, never overwriting an existing image. Only accepts detected chessboard images.

Parameters

- **camera** (*BaslerCamera*) – camera object
- **dirname** (*str*) – directory to save the images

`src.calibration.camera_charuco_calibration_routine.initialise_camera()` → BaslerCamera

initialises the camera and returns the camera object

Returns
camera object

Return type
BaslerCamera

`src.calibration.camera_charuco_calibration_routine.main()`

src.calibration.hand_eye_calib_from_data module

`src.calibration.hand_eye_calib_from_data.load_json_data(`*filename: str*`)` → List[List[*SE3*]]

Load the json data from the file

Parameters
filename (*str*) – filename to load the data from

Returns
list of robot positions and eye positions

Return type
List[List[*SE3*]]

`src.calibration.hand_eye_calib_from_data.main()`

`src.calibration.hand_eye_calib_from_data.solve_AX_YB(`*a*, *b*`)`

Solve A^iX=YB^i, return X, Y :param a: list of SE3 objects :param b: list of SE3 objects

Returns
SE3 objects

Return type
X, Y

src.calibration.hand_eye_calib_with_moves module

`src.calibration.hand_eye_calib_with_moves.find_aruco_id_vec(`*corners*, *ids*, *ARUCO_SIZE: int*`)` → *SE3*

Find the vectors of the wanted aruco id in the image.

Parameters

- **corners** (*list*) – list of aruco corners
- **ids** (*list*) – list of aruco ids
- **ARUCO_SIZE** (*int*) – size of the aruco marker

Returns
SE3 object of the aruco marker

Return type
SE3

`src.calibration.hand_eye_calib_with_moves.generate_robot_positions(`*robot:* Robot`)`

Generate a list of robot positions (in)

Parameters
robot (Robot) – robot object

Returns
list of robot positions

Return type
list

`src.calibration.hand_eye_calib_with_moves.`**`generate_robot_positions_new`**`(`*robot: Robot*`)` → List

Generate a list of robot positions

Parameters
`robot` (`Robot`) – robot object

Returns
list of robot positions

Return type
List

`src.calibration.hand_eye_calib_with_moves.`**`hand_eye_take_image`**`(`*dirname: str*`)` → List

Cycles through robot positions and takes images at each position. Format of robot positions is a list of robot SE3s.

Parameters
`dirname` (*`str`*) – directory to save the images

Returns
list of SE3s of the robot

Return type
list

`src.calibration.hand_eye_calib_with_moves.`**`initialise_camera`**`()` → BaslerCamera

initialises the camera and returns the camera object

Returns
camera object

Return type
BaslerCamera

`src.calibration.hand_eye_calib_with_moves.`**`main`**`()`

`src.calibration.hand_eye_calib_with_moves.`**`solve_AX_YB`**`(`*a*, *b*`)`

Solve A^iX=YB^i, return X, Y

Parameters

- **`a`** – list of SE3 objects
- **`b`** – list of SE3 objects

Returns
SE3 objects

Return type
X, Y

Module contents

calibration scripts for camera and hand-eye calibration.

src.move_commands package

Submodules

src.move_commands.go_home module

src.move_commands.move module

src.move_commands.move_outside_cam module

src.move_commands.move_to_gate module

src.move_commands.release module

Module contents

Programs for easier manipulation of the robot.

1.2.2 Module contents

source file for the project.

1.3 tests package

1.3.1 Submodules

1.3.2 tests.aruco_test module

Gets the camera image and detects the Aruco markers on it.

tests.aruco_test.**main**()

1.3.3 tests.camera_test module

Program to display one photo of the camera

tests.camera_test.**main**()

1.3.4 tests.conseq_moves module

First attempt at creating consequences of moves.

tests.conseq_moves.**main**()

1.3.5 tests.conseq_moves_2 module

First attempt at creating consequences of moves.

tests.conseq_moves_2.**main**()

1.3.6 tests.conseq_moves_3 module

First attempt at creating consequences of moves.

tests.conseq_moves_3.**main**()

1.3.7 tests.conseq_moves_4 module

First attempt at creating consequences of moves.

`tests.conseq_moves_4.`**`main`**`()`

1.3.8 tests.conseq_moves_5 module

First attempt at creating consequences of moves.

`tests.conseq_moves_5.`**`main`**`()`

1.3.9 tests.reproj_error module

Test to check the reprojection error of the camera model.

`tests.reproj_error.`**`main`**`()`

1.3.10 tests.robot_wrapper_test module

`tests.robot_wrapper_test.`**`generate_robot_positions`**`(`*robot:* Robot`)`

Generate a list of robot positions (in)

`tests.robot_wrapper_test.`**`main`**`()`

1.3.11 tests.test_robot2target module

Gets the camera image and detects the Aruco markers on it.

`tests.test_robot2target.`**`main`**`()`

1.3.12 tests.test_robot_camera_calib module

`tests.test_robot_camera_calib.`**`find_aruco_id_vec`**`(`*corners*, *ids*, *ARUCO_SIZE: int*, *wanted_id: int*`)` → *SE3*

Find the vectors of the wanted aruco id in the image.

`tests.test_robot_camera_calib.`**`main`**`()`

1.3.13 tests.test_show_positions module

1.3.14 Module contents

Test files for the camera calibration module, hand-eye calibration module and robot movement modules.

PYTHON MODULE INDEX

INDEX