skrobot Documentation

Iori Yanokura, Kentaro Wada

Apr 20, 2023

CONTENTS

Scikit-Robot is a simple pure-Python library for loading, manipulating, and visualizing URDF files and robot specifications. For example, here's a rendering of a PR2 robot moving after being loaded by this library.

CHAPTER

ONE

INSTALLATION AND SETUP GUIDE

1.1 Python Installation

This package is pip-installable for any Python version. Simply run the following command:

```
pip install scikit-robot
```

1.2 Installing in Development Mode

If you're planning on contributing to this repository, please see the *Development Guide*.

CHAPTER

TWO

USAGE EXAMPLES

This page documents several simple use cases for you to try out. For full details, see the *API Reference*, and check out the full class reference for *RobotModel*.

2.1 Loading from a File

You can load a URDF from any `.urdf` file, as long as you fix the links to be relative or absolute links rather than ROS resource URLs.

```
>>> import skrobot
>>> robot_model = skrobot.models.urdf.RobotModelFromURDF(urdf_file=skrobot.data.pr2_
↪urdfpath())
```

2.2 Visualization

```
>>> viewer = skrobot.viewers.TrimeshSceneViewer(resolution=(640, 480))
>>> viewer.add(robot_model)
>>> viewer.show()
```

If you would like to update renderer:

```
>>> viewer.redraw()
```

2.3 Accessing Links and Joints

You have direct access to link and joint information.

```
>>> for link in robot_model.link_list:
...     print(link.name)
```

```
>>> for joint in robot_model.joint_list:
...     print(joint.name)
```

```
>>> robot_model.l_elbow_flex_joint.joint_angle()
0.0
```

```
>>> robot_model.l_elbow_flex_joint.joint_angle(-0.5)
-0.5
```

```
>>> robot_model.l_elbow_flex_joint.joint_angle()
-0.5
```

2.4 Inverse Kinematics

First, set the initial pose. Note that the position of the prismatic joint is in [m] and angles of rotational joints are in [rad].

```
>>> robot_model.torso_lift_joint.joint_angle(0.05)
>>> robot_model.l_shoulder_pan_joint.joint_angle(60 * 3.14/180.0)
>>> robot_model.l_shoulder_lift_joint.joint_angle(74 * 3.14/180.0)
>>> robot_model.l_upper_arm_roll_joint.joint_angle(70* 3.14/180.0)
>>> robot_model.l_elbow_flex_joint.joint_angle(-120 * 3.14/180.0)
>>> robot_model.l_forearm_roll_joint.joint_angle(20 * 3.14/180.0)
>>> robot_model.l_wrist_flex_joint.joint_angle(-30 * 3.14/180.0)
>>> robot_model.l_wrist_roll_joint.joint_angle(180 * 3.14/180.0)
>>> robot_model.r_shoulder_pan_joint.joint_angle(-60 * 3.14/180.0)
>>> robot_model.r_shoulder_lift_joint.joint_angle(74 * 3.14/180.0)
>>> robot_model.r_upper_arm_roll_joint.joint_angle(-70 * 3.14/180.0)
>>> robot_model.r_elbow_flex_joint.joint_angle(-120 * 3.14/180.0)
>>> robot_model.r_forearm_roll_joint.joint_angle(-20 * 3.14/180.0)
>>> robot_model.r_wrist_flex_joint.joint_angle(-30 * 3.14/180.0)
>>> robot_model.r_wrist_roll_joint.joint_angle(180 * 3.14/180.0)
>>> robot_model.head_pan_joint.joint_angle(0)
>>> robot_model.head_tilt_joint.joint_angle(0)
```

Next, set move_target and link_list

```
>>> rarm_end_coords = skrobot.coordinates.CascadedCoords(
...             parent=robot_model.r_gripper_tool_frame,
...             name='rarm_end_coords')
>>> move_target = rarm_end_coords
>>> link_list = [
...     robot_model.r_shoulder_pan_link,
...     robot_model.r_shoulder_lift_link,
...     robot_model.r_upper_arm_roll_link,
...     robot_model.r_elbow_flex_link,
...     robot_model.r_forearm_roll_link,
...     robot_model.r_wrist_flex_link,
...     robot_model.r_wrist_roll_link]
```

Set target_coords.

```
>>> target_coords = skrobot.coordinates.Coordinates([0.5, -0.3, 0.7], [0, 0, 0])
>>> robot_model.inverse_kinematics(
...     target_coords,
...     link_list=link_list,
...     move_target=move_target)
```

CHAPTER

THREE

API REFERENCE

3.1 Coordinates

3.1.1 Coordinates classes

skrobot.coordinates.Coordinates	Coordinates class to manipulate rotation and translation.
skrobot.coordinates.CascadedCoords	

skrobot.coordinates.Coordinates

class skrobot.coordinates.**Coordinates**(*pos=None*, *rot=None*, *name=None*, *hook=None*, *check_validity=True*)

Coordinates class to manipulate rotation and translation.

Parameters

- **pos** (*list or numpy.ndarray or None*) – shape of (3,) translation vector. or 4x4 homogeneous transformation matrix. If the homogeneous transformation matrix is given, *rot* will be overwritten. If this value is *None*, set [0, 0, 0] vector as default.
- **rot** (*list or numpy.ndarray or None*) – we can take 3x3 rotation matrix or [yaw, pitch, roll] or quaternion [w, x, y, z] order If this value is *None*, set the identity matrix as default.
- **name** (*str or None*) – name of this coordinates
- **check_validity** (*bool (optional)*) – Default *True*. If this value is *True*, check whether an input rotation and an input translation are valid.

Methods

T()

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

axis(*ax*)

changed()

Return False

This is used for CascadedCoords compatibility

Returns
False – always return False

Return type
bool

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xy’, ‘yz’, ‘zx’, ‘xx’, ‘yy’, ‘zz’, True or False(None).

Returns
dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...           pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 2.0, 'y')
>>> c1.difference_position(c2)
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xx’, ‘yy’, ‘zz’, ‘xm’, ‘ym’, ‘zm’, ‘xy’, ‘yx’, ‘yz’, ‘zy’, ‘zx’, ‘xz’, True or False(None).

Returns
dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option ['xm', 'ym', 'zm'], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

get_transform()

Return Transform object

Returns
transform – correnssponding Transform to this coordinates

Return type
skrobot.coordinates.base.Transform

inverse_rotate_vector(*v*)

inverse_transform_vector(*vec*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or skrobot.coordinates.Coordinates*) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.

- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

newcoords(*c*, *pos=None*, *check_validity=True*)

Update of coords is always done through newcoords.

Parameters

- **c** (skrobot.coordinates.Coordinates *or* numpy.ndarray) – If pos is *None*, *c* means new Coordinates. If pos is given, *c* means rotation matrix.
- **pos** (numpy.ndarray *or None*) – new translation.
- **check_validity** (*bool*) – If this value is *True*, check whether an input rotation and an input translation are valid.

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (*numpy.ndarray*) – 3x3 rotation matrix.
- **wrt** (*str or* skrobot.coordinates.Coordinates) – reference coordinates.

parent_orientation(*v*, *wrt*)

rotate(*theta*, *axis=None*, *wrt='local'*)

Rotate this coordinate by given theta and axis.

This coordinate system is rotated relative to theta radians around the *axis* axis. Note that this function does not change a position of this coordinate. If you want to rotate this coordinates around with world frame, you can use *transform* function. Please see examples.

Parameters

- **theta** (*float*) – relartive rotation angle in radian.
- **axis** (*str or None or numpy.ndarray*) – axis of rotation. The value of *axis* is represented as *wrt* frame.
- **wrt** (*str or* skrobot.coordinates.Coordinates) –

Returns
self

Return type
skrobot.coordinates.Coordinates

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates()
>>> c.translate((1.0, 0, 0))
>>> c.rotate(pi / 2.0, 'z', wrt='local')
>>> c.translation
array([1., 0., 0.])
```

```
>>> c.transform(Coordinates().rotate(np.pi / 2.0, 'z'), wrt='world')
>>> c.translation
array([0., 1., 0.])
```

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (*numpy.ndarray*) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*mat*, *wrt='local'*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

transform(*c*, *wrt='local'*, *out=None*)

Transform this coordinates by coords based on wrt

Note that this function changes this coordinates translation and rotation. If you would like not to change this coordinates, Please use copy_worldcoords() or give *out*.

Parameters

- **c** (skrobot.coordinates.Coordinates) – coordinate
- **wrt** (*str or* skrobot.coordinates.Coordinates) – If wrt is 'local' or self, multiply c from the right. If wrt is 'world' or 'parent' or self.parent, transform c with respect to worldcoord. If wrt is Coordinates, transform c with respect to c.
- **out** (*None or* skrobot.coordinates.Coordinates) – If the *out* is specified, set new coordinates to *out*. Note that if the *out* is given, these coordinates don't change.

Returns
self – return this coordinate

Return type
skrobot.coordinates.Coordinates

Examples

transform_vector(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

transformation(*c2*, *wrt='local'*)

translate(*vec*, *wrt='local'*)

Translate this coordinates.

Note that this function changes this coordinates self. So if you don't want to change this class, use copy_worldcoords()

Parameters

- **vec** (*list or numpy.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*str or Coordinates (optional)*) – translate with respect to wrt.

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.], dtype=float32)
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3], dtype=float32)
```

```
>>> c = Coordinates()
>>> c.copy_worldcoords().translate([0.1, 0.2, 0.3])
>>> c.translation
array([0., 0., 0.], dtype=float32)
```

```
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([ 0.3,  0.2, -0.1])
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3], 'world')
>>> c.translation
array([0.1, 0.2, 0.3])
```

worldcoords()

Return thisself

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns
self.translation – translation of this coordinate

Return type
numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns
self.rotation – rotation matrix of this coordinate

Return type
numpy.ndarray

__eq__(*value*, /)

Return self==value.

__ne__(*value*, /)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, /)

Return self>=value.

__mul__(*other_c*)

Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

__pow__(*exponent*)

Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

dimension

Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

dual_quaternion

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

Return type
numpy.ndarray

z_axis

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray

skrobot.coordinates.CascadedCoords

`class skrobot.coordinates.CascadedCoords(parent=None, *args, **kwargs)`

Methods

`T()`

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

`assoc(child, relative_coords='world', force=False, **kwargs)`

Associate child coords to this coordinate system.

If *relative_coords* is *None* or 'world', the translation and rotation of childcoords in the world coordinate system do not change. If *relative_coords* is specified, childcoords is assoced at translation and rotation of *relative_coords*. By default, if child is already assoced to some other coords, raise an exception. But if *force* is *True*, you can overwrite the existing assoc relation.

Parameters

- **child** (CascadedCoords) – child coordinates.

- **relative_coords** (*None or Coordinates or str*) – child coordinates' relative coordinates.
- **force** (*bool*) – predicate for overwriting the existing assoc-relation

Returns
child – assoced child.

Return type
CascadedCoords

Examples

```
>>> from skrobot.coordinates import CascadedCoords
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords)
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
>>> child_coords.translation
array([0., 1., 0.])
```

None and 'world' have the same meaning.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='world')
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
```

If *relative_coords* is 'local', *child* is associated at world translation and world rotation of *child* from this coordinate system.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='local')
>>> child_coords.worldpos()
array([2., 1., 0.])
>>> child_coords.translation
array([1., 1., 0.])
```

axis(*ax*)

changed()

Return False

This is used for CascadedCoords compatibility

Returns
False – always return False

Return type
bool

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take 'x', 'y', 'z', 'xy', 'yz', 'zx', 'xx', 'yy', 'zz', True or False(None).

Returns

dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type

numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...          pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 2.0, 'y')
>>> c1.difference_position(c2)
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take 'x', 'y', 'z', 'xx', 'yy', 'zz', 'xm', 'ym', 'zm', 'xy', 'yx', 'yz', 'zy', 'zx', 'xz', True or False(None).

Returns

dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option [‘xm’, ‘ym’, ‘zm’], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

dissoc(*child*)

get_transform()

Return Transform object

Returns
transform – corrensponding Transform to this coordinates

Return type
skrobot.coordinates.base.Transform

inverse_rotate_vector(*v*)

inverse_transform_vector(*v*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or skrobot.coordinates.Coordinates*) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.
- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

newcoords(*c*, *pos=None*, *check_validity=True*)

Update this coordinates.

This function records that this CascadedCoords has changed and recursively records the change to descendants of this CascadedCoords.

Parameters

- **c** (skrobot.coordinates.Coordinates *or* numpy.ndarray) – If pos is *None*, *c* means new Coordinates. If pos is given, *c* means rotation matrix.
- **pos** (numpy.ndarray *or None*) – new translation.
- **check_validity** (*bool*) – If this value is *True*, check whether an input rotation and an input translation are valid.

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (*numpy.ndarray*) – 3x3 rotation matrix.
- **wrt** (*str or* skrobot.coordinates.Coordinates) – reference coordinates.

parent_orientation(*v*, *wrt*)

parentcoords()

rotate(*theta*, *axis*, *wrt='local'*)

Rotate this coordinate.

Rotate this coordinate relative to axis by theta radians with respect to wrt.

Parameters

- **theta** (*float*) – radian
- **axis** (*str or numpy.ndarray*) – ‘x’, ‘y’, ‘z’ or vector
- **wrt** (*str or Coordinates*) –

Return type
self

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (*numpy.ndarray*) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*matrix*, *wrt*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

transform(*c*, *wrt='local'*, *out=None*)

Transform this coordinates

Parameters

- **c** (skrobot.coordinates.Coordinates) – coordinates
- **wrt** (*str or* skrobot.coordinates.Coordinates) – transform this coordinates with respect to wrt. If wrt is 'local' or self, multiply c from the right. If wrt is 'parent' or self.parent, transform c with respect to parentcoords. (multiply c from the left.) If wrt is Coordinates, transform c with respect to c.
- **out** (*None or* skrobot.coordinates.Coordinates) – If the *out* is specified, set new coordinates to *out*. Note that if the *out* is given, these coordinates don't change.

Returns
self – return self

Return type
skrobot.coordinates.CascadedCoords

transform_vector(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

transformation(*c2*, *wrt='local'*)

translate(*vec*, *wrt='local'*)

Translate this coordinates.

Note that this function changes this coordinates self. So if you don't want to change this class, use copy_worldcoords()

Parameters

- **vec** (*list or numpy.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*str or* Coordinates *(optional)*) – translate with respect to wrt.

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.], dtype=float32)
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3], dtype=float32)
```

```
>>> c = Coordinates()
>>> c.copy_worldcoords().translate([0.1, 0.2, 0.3])
>>> c.translation
array([0., 0., 0.], dtype=float32)
```

```
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([ 0.3,  0.2, -0.1])
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3], 'world')
>>> c.translation
array([0.1, 0.2, 0.3])
```

update(*force=False*)

worldcoords()

Calculate rotation and position in the world.

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns
self.translation – translation of this coordinate

Return type
numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns
self.rotation – rotation matrix of this coordinate

Return type
numpy.ndarray

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, /)

Return self>=value.

__mul__(*other_c*)

Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

__pow__(*exponent*)

Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

descendants

dimension

Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

dual_quaternion

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

parent

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

(continues on next page)

(continued from previous page)

```
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

Return type
numpy.ndarray

z_axis

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray

3.1.2 Coordinates utilities

skrobot.coordinates.geo.midcoords	Returns mid (or p) coordinates of given two coordinates c1 and c2.
skrobot.coordinates.geo.orient_coords_to_axis	Orient axis to the direction
skrobot.coordinates.base.random_coords	Return Coordinates class has random translation and rotation
skrobot.coordinates.base.coordinates_distance	
skrobot.coordinates.base.transform_coords	Return Coordinates by applying c1 to c2 from the left

skrobot.coordinates.geo.midcoords

skrobot.coordinates.geo.**midcoords**(*p*, *c1*, *c2*)

Returns mid (or p) coordinates of given two coordinates c1 and c2.

Parameters

- **p** (*float*) – ratio of c1:c2
- **c1** (skrobot.coordinates.Coordinates) – Coordinates
- **c2** (skrobot.coordinates.Coordinates) – Coordinates

Returns

coordinates – midcoords

Return type

skrobot.coordinates.Coordinates

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.geo import midcoords
>>> c1 = Coordinates()
>>> c2 = Coordinates(pos=[0.1, 0, 0])
>>> c = midcoords(0.5, c1, c2)
>>> c.translation
array([0.05, 0.  , 0.  ])
```

skrobot.coordinates.geo.orient_coords_to_axis

skrobot.coordinates.geo.**orient_coords_to_axis**(*target_coords*, *v*, *axis='z'*, *eps=0.005*)

Orient axis to the direction

Orient axis in target_coords to the direction specified by v.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) –
- **v** (*list or numpy.ndarray*) – position of target [x, y, z]
- **axis** (*list or string or numpy.ndarray*) – see _wrap_axis function

- **eps** (*float (optional)*) – eps

Returns
target_coords

Return type
skrobot.coordinates.Coordinates

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.geo import orient_coords_to_axis
>>> c = Coordinates()
>>> oriented_coords = orient_coords_to_axis(c, [1, 0, 0])
>>> oriented_coords.translation
array([0., 0., 0.])
>>> oriented_coords.rpy_angle()
(array([0.        , 1.57079633, 0.        ]),
 array([3.14159265, 1.57079633, 3.14159265]))
```

```
>>> c = Coordinates(pos=[0, 1, 0])
>>> oriented_coords = orient_coords_to_axis(c, [0, 1, 0])
>>> oriented_coords.translation
array([0., 1., 0.])
>>> oriented_coords.rpy_angle()
(array([ 0.        , -0.        , -1.57079633]),
 array([ 3.14159265, -3.14159265,  1.57079633]))
```

```
>>> c = Coordinates(pos=[0, 1, 0]).rotate(np.pi / 3, 'y')
>>> oriented_coords = orient_coords_to_axis(c, [0, 1, 0])
>>> oriented_coords.translation
array([0., 1., 0.])
>>> oriented_coords.rpy_angle()
(array([-5.15256299e-17,  1.04719755e+00, -1.57079633e+00]),
 array([3.14159265, 2.0943951 , 1.57079633]))
```

skrobot.coordinates.base.random_coords

skrobot.coordinates.base.**random_coords**()

Return Coordinates class has random translation and rotation

skrobot.coordinates.base.coordinates_distance

skrobot.coordinates.base.**coordinates_distance**(*c1*, *c2*, *c=None*)

skrobot.coordinates.base.transform_coords

skrobot.coordinates.base.**transform_coords**(*c1*, *c2*, *out=None*)

Return Coordinates by applying c1 to c2 from the left

Parameters

- **c1** (skrobot.coordinates.Coordinates) –
- **c2** (skrobot.coordinates.Coordinates) – Coordinates
- **c3** (skrobot.coordinates.Coordinates *or None*) – Output argument. If this value is specified, the results will be in-placed.

Returns

Coordinates(pos=translation, rot=q) – new coordinates

Return type

skrobot.coordinates.Coordinates

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates()
>>> c2 = Coordinates()
>>> c3 = transform_coords(c1, c2)
>>> c3.translation
array([0., 0., 0.])
>>> c3.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(pi / 2.0, 'y')
>>> c3 = transform_coords(c1, c2)
>>> c3.translation
array([ 0.4       , -0.03660254,  0.09019238])
>>> c3.rotation
>>> c3.rotation
array([[ 1.94289029e-16,  0.00000000e+00,  1.00000000e+00],
       [ 8.66025404e-01,  5.00000000e-01, -1.66533454e-16],
       [-5.00000000e-01,  8.66025404e-01,  2.77555756e-17]])
```

3.1.3 Quaternion Classes

skrobot.coordinates.quaternion.Quaternion	Class for handling Quaternion.
skrobot.coordinates.dual_quaternion.DualQuaternion	Class for handling dual quaternions and their interpolations.

skrobot.coordinates.quaternion.Quaternion

class skrobot.coordinates.quaternion.**Quaternion**(*w=1.0*, *x=0.0*, *y=0.0*, *z=0.0*, *q=None*)

Class for handling Quaternion.

Parameters

- **w** (*float or numpy.ndarray*) –
- **x** (*float*) –
- **y** (*float*) –
- **z** (*float*) –
- **q** (*None or numpy.ndarray*) – if q is not specified, use w, x, y, z.

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q
#<Quaternion 0x1283bde48 w: 1.0 x: 0.0 y: 0.0 z: 0.0>
>>> q = Quaternion([1, 2, 3, 4])
>>> q
#<Quaternion 0x1283cad68 w: 1.0 x: 2.0 y: 3.0 z: 4.0>
>>> q = Quaternion(q=[1, 2, 3, 4])
>>> q
#<Quaternion 0x1283bd438 w: 1.0 x: 2.0 y: 3.0 z: 4.0>
>>> q = Quaternion(1, 2, 3, 4)
>>> q
#<Quaternion 0x128400198 w: 1.0 x: 2.0 y: 3.0 z: 4.0>
>>> q = Quaternion(w=0.0, x=1.0, y=0.0, z=0.0)
>>> q
#<Quaternion 0x1283cc2e8 w: 0.0 x: 1.0 y: 0.0 z: 0.0>
```

Methods

T()

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> q.q = [1, 2, 3, 4]
>>> q.T()
array([[-0.66666667,  0.13333333,  0.73333333,  0.        ],
       [ 0.66666667, -0.33333333,  0.66666667,  0.        ],
       [ 0.33333333,  0.93333333,  0.13333333,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  1.        ]])
```

copy()

Return copy of this Quaternion

Returns

Quaternion(q=self.q.copy()) – copy of this quaternion

Return type

skrobot.coordinates.quaternion.Quaternion

normalize()

Normalize this quaternion.

Note that this function changes wxyz property.

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion([1, 2, 3, 4])
>>> q.q
array([1., 2., 3., 4.])
>>> q.normalize()
>>> q.q
array([0.18257419, 0.36514837, 0.54772256, 0.73029674])
```

__eq__(*value*, /)

Return self==value.

__ne__(*value*, /)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, */*)
Return self>=value.

__neg__()

__add__(*cls*)

__sub__(*cls*)

__mul__(*cls*)

__rmul__(*cls*)

__div__(*cls*)

__truediv__(*cls*)

Attributes

angle
Return rotation angle of this quaternion

Returns
theta – rotation angle with respect to self.axis

Return type
float

axis
Return axis of this quaternion.

Note that this property return normalized axis.

Returns
axis – normalized axis vector

Return type
numpy.ndarray

conjugate
Return conjugate of this quaternion

Returns
Quaternion – new Quaternion class has this quaternion's conjugate

Return type
skrobot.coordinates.quaternion.Quaternion

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q.conjugate
#<Quaternion 0x12f2dfb38 w: 1.0 x: -0.0 y: -0.0 z: -0.0>
>>> q.q = [0, 1, 0, 0]
>>> q.conjugate
#<Quaternion 0x12f303c88 w: 0.0 x: -1.0 y: -0.0 z: -0.0>
```

inverse

Return inverse of this quaternion

Returns
q – new Quaternion class has inverse of this quaternion

Return type
skrobot.coordinates.quaternion.Quaternion

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q
#<Quaternion 0x127e6da58 w: 1.0 x: 0.0 y: 0.0 z: 0.0>
>>> q.inverse
#<Quaternion 0x1281bbda0 w: 1.0 x: -0.0 y: -0.0 z: -0.0>
>>> q.q = [0, 1, 0, 0]
>>> q.inverse
#<Quaternion 0x1282b0cc0 w: 0.0 x: -1.0 y: -0.0 z: -0.0>
```

norm

Return norm of this quaternion

Returns
quaternion_norm(self.q) – norm of this quaternion

Return type
float

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q.norm
1.0
>>> q = Quaternion([1, 2, 3, 4])
>>> q.norm
5.477225575051661
>>> q.normalized.norm
0.9999999999999999
```

normalized

Return Normalized quaternion.

Returns
normalized quaternion – return quaternion which is norm == 1.0.

Return type
skrobot.coordinates.quaternion.Quaternion

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion([1, 2, 3, 4])
>>> normalized_q = q.normalized
>>> normalized_q.q
array([0.18257419, 0.36514837, 0.54772256, 0.73029674])
>>> q.q
array([1., 2., 3., 4.])
```

q

Return quaternion

Returns
self._q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q.q
array([1., 0., 0., 0.])
>>> q = Quaternion(w=0.0, x=1.0, y=0.0, z=0.0)
>>> q.q
array([0., 1., 0., 0.])
```

rotation

Return rotation matrix.

Note that this property internally normalizes quaternion.

Returns
quaternion2matrix(self.q) – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.quaternion import Quaternion
>>> q = Quaternion()
>>> q
#<Quaternion 0x12f1aa6a0 w: 1.0 x: 0.0 y: 0.0 z: 0.0>
>>> q.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> q.q = [0, 1, 0, 0]
>>> q
#<Quaternion 0x12f1aa6a0 w: 0.0 x: 1.0 y: 0.0 z: 0.0>
```

(continues on next page)

(continued from previous page)

```
>>> q.rotation
array([[ 1.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0., -1.]])
>>> q.q = [1, 2, 3, 4]
>>> q
#<Quaternion 0x12f1aa6a0 w: 1.0 x: 2.0 y: 3.0 z: 4.0>
>>> q.rotation
array([[-0.66666667,  0.13333333,  0.73333333],
       [ 0.66666667, -0.33333333,  0.66666667],
       [ 0.33333333,  0.93333333,  0.13333333]])
```

w

Return w element

Returns
self.q[0] – w element of this quaternion

Return type
float

x

Return x element

Returns
self.q[1] – x element of this quaternion

Return type
float

xyz

Return xyz vector of this quaternion

Returns
quaternion_xyz – xyz elements of this quaternion

Return type
numpy.ndarray

y

Return y element

Returns
self.q[2] – y element of this quaternion

Return type
float

z

Return z element

Returns
self.q[3] – z element of this quaternion

Return type
float

skrobot.coordinates.dual_quaternion.DualQuaternion

class skrobot.coordinates.dual_quaternion.**DualQuaternion**(*qr=[1, 0, 0, 0]*, *qd=[0, 0, 0, 0]*, *enforce_unit_norm=False*)

Class for handling dual quaternions and their interpolations.

Parameters

- **qr** (*list or numpy.ndarray*) –
- **qd** (*list or numpy.ndarray*) – element of dual quaternion
- **enforce_unit_norm** (*bool (optional)*) – if True, norm should be 1.0.

Methods

T()

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.dual_quaternion import DualQuaternion
>>> dq = DualQuaternion()
>>> dq.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> dq = Coordinates().rotate(pi / 2.0, 'y').        ...                    ␣
↪translate((0.1, 0.2, 0.3)).        ...                    dual_quaternion
>>> dq.T()
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         3.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
        -1.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

copy()

Return a copy of this quaternion.

Returns
copied DualQuaternion instance

Return type
DualQuaternion

difference_position(*other_dq*)

Return difference position

Parameters
other_dq (skrobot.coordinates.dual_quaternion.DualQuaternion) – dual quaternion

Returns
dif_pos – difference position's norm

Return type
float

difference_rotation(*other_dq*)

Return difference rotation distance

Parameters
other_dq (skrobot.coordinates.dual_quaternion.DualQuaternion) – dual quaternion

Returns
dif_rot – angle distance in radian.

Return type
float

enforce_positive_q_rot_w()

static interpolate(*dq0*, *dq1*, *t*)

Return interpolated dual quaternion

Parameters

- **dq0** (skrobot.coordinates.dual_quaternion.DualQuaternion) –
- **dq1** (skrobot.coordinates.dual_quaternion.DualQuaternion) – dual quaternion
- **t** (*float*) – ratio of interpolation. Must be 0 <= t <= 1.0.

normalize()

Normalize this dual quaternion

Note that this function changes property.

Returns
self – return self

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

pose()

Return [x, y, z, wx, wy, wz, wq] elements.

Returns
pose – [x, y, z, wx, wy, wz, wq] pose

Return type
numpy.ndarray

screw_axis()

Return screw axis

Calculates rotation, translation and screw axis from dual quaternion.

Returns
screw_axis, theta, translation – screw axis of this dual quaternion. rotation angle in radian. translation

Return type
tuple(numpy.ndarray, float, float)

`__eq__`(*value*, /)
Return self==value.

`__ne__`(*value*, /)
Return self!=value.

`__lt__`(*value*, /)
Return self<value.

`__le__`(*value*, /)
Return self<=value.

`__gt__`(*value*, /)
Return self>value.

`__ge__`(*value*, /)
Return self>=value.

`__add__`(*val*)

`__mul__`(*val*)

`__rmul__`(*val*)

Attributes

`angle`
Return rotation angle of this dual quaternion

Returns
self.qr.angle – this dual quaternion's rotation angle with respect to self.axis. See skrobot.coordinates.quaternion.Quaternion.angle.

Return type
float

`axis`
Return axis of this dual quaternion

Returns
self.qr.axis – this dual quaternion's axis. See See skrobot.coordinates.quaternion.Quaternion.axis.

Return type
numpy.ndarray

`conjugate`
Return conjugate of this dual quaternion

Returns
DualQuaternion – new DualQuaternion class has this dual quaternion's conjugate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

dq

Return flatten vector of this dual quaternion

Returns
np.concatenate([self.qr.q, self.qd.q]) – (1x8) vector of this dual quaternion

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.dual_quaternion import DualQuaternion
>>> dq = DualQuaternion()
>>> dq.dq
array([1., 0., 0., 0., 0., 0., 0., 0.])
```

inverse

Return inverse of this dual quaternion

Returns
dq – new DualQuaternion class has inverse of this dual quaternion

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

norm

Return pair of norm of this dual quaternion

Returns
qr_norm, qd_norm – qr and qd's norm

Return type
tuple(float, float)

Examples

```
>>> from skrobot.coordinates.dual_quaternion import DualQuaternion
>>> dq = DualQuaternion()
>>> dq.norm
(1.0, 0.0)
```

normalized

Return normalized this dual quaternion

Returns
skrobot.coordinates.dual_quaternion.DualQuaternion normalized dual quaternion

Return type
DualQuaternion(qr, qd, True)

qd

Return translation quaternion

Returns
self._qd – quaternion indicating translation

Return type
skrobot.coordinates.quaternion.Quaternion

`qr`

Return orientation

Returns
self._qr – [w, x, y, z] order

Return type
numpy.narray

`quaternion`

Return this dual quaternion's qr (rotation)

Returns
dq.qr – rotation quaternion

Return type
skrobot.coordinates.quaternion.Quaternion

`rotation`

Return rotation matrix of this dual quaternion

Returns
dq.qr.rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.dual_quaternion.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.dual_quaternion.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

`scalar`

The scalar part of the dual quaternion.

Returns
scalar – scalar

Return type
float

translation

Return translation of this dual quaternion.

Returns
q_translation.xyz – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.dual_quaternion.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.dual_quaternion.translation
array([0.1, 0.2, 0.3])
```

3.2 Models

3.2.1 Robot Model class

skrobot.model.RobotModel

skrobot.model.RobotModel

class skrobot.model.**RobotModel**(*link_list=None*, *joint_list=None*, *root_link=None*)

Methods

T()

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

angle_vector(*av=None*, *return_av=None*)

Returns angle vector

If av is given, it updates angles of all joint. If given av violate min/max range, the value is modified.

assoc(*child*, *relative_coords='world'*, *force=False*, ***kwargs*)

Associate child coords to this coordinate system.

If *relative_coords* is *None* or 'world', the translation and rotation of childcoords in the world coordinate system do not change. If *relative_coords* is specified, childcoords is assoced at translation and rotation of *relative_coords*. By default, if child is already assoced to some other coords, raise an exception. But if *force* is *True*, you can overwrite the existing assoc relation.

Parameters

- **child** (CascadedCoords) – child coordinates.
- **relative_coords** (*None or* Coordinates *or str*) – child coordinates' relative coordinates.
- **force** (*bool*) – predicate for overwriting the existing assoc-relation

Returns

child – assoced child.

Return type

CascadedCoords

Examples

```
>>> from skrobot.coordinates import CascadedCoords
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords)
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
>>> child_coords.translation
array([0., 1., 0.])
```

None and 'world' have the same meaning.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='world')
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
```

If *relative_coords* is 'local', *child* is associated at world translation and world rotation of *child* from this coordinate system.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='local')
>>> child_coords.worldpos()
array([2., 1., 0.])
>>> child_coords.translation
array([1., 1., 0.])
```

axis(*ax*)

calc_inverse_jacobian(*jacobi*, *manipulability_limit=0.1*, *manipulability_gain=0.001*, *weight=None*, **args*, ***kwargs*)

calc_inverse_kinematics_nspace_from_link_list(*link_list*, *avoid_nspace_gain=0.01*, *union_link_list=None*, *n_joint_dimension=None*, *null_space=None*, *additional_nspace_list=None*, *weight=None*)

calc_inverse_kinematics_weight_from_link_list(*link_list*, *avoid_weight_gain=1.0*, *union_link_list=None*, *n_joint_dimension=None*, *weight=None*, *additional_weight_list=[]*)

Calculate all weight from link list.

calc_jacobian_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

calc_jacobian_from_link_list(*move_target*, *link_list=None*, *transform_coords=None*, *rotation_axis=None*, *translation_axis=None*, *col_offset=0*, *dim=None*, *jacobian=None*, *additional_jacobi_dimension=0*, *n_joint_dimension=None*, **args*, ***kwargs*)

calc_joint_angle_from_min_max_table(*index*, *j*, *av*)

calc_joint_angle_speed(*union_vel*, *angle_speed=None*, *angle_speed_blending=0.5*, *jacobi=None*, *j_sharp=None*, *null_space=None*, **args*, ***kwargs*)

calc_joint_angle_speed_gain(*union_link_list*, *dav*, *periodic_time*)

calc_nspace_from_joint_limit(*avoid_nspace_gain*, *union_link_list*, *weight*)

Calculate null-space according to joint limit.

calc_target_axis_dimension(*rotation_axis*, *translation_axis*)

rotation-axis, translation-axis -> both list and atom OK.

calc_target_joint_dimension(*link_list*)

calc_union_link_list(*link_list*)

calc_vel_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

Calculate 0 velocity for keeping interlocking joint.

at the same joint angle.

calc_vel_from_pos(*dif_pos*, *translation_axis*, *p_limit=100.0*)

Calculate velocity from difference position

Parameters

- **dif_pos** (*np.ndarray*) – [m] order
- **translation_axis** (*str*) – see calc_dif_with_axis

Returns
vel_p

Return type
np.ndarray

calc_vel_from_rot(*dif_rot*, *rotation_axis*, *r_limit=0.5*)

calc_weight_from_joint_limit(*avoid_weight_gain*, *link_list*, *union_link_list*, *weight*, *n_joint_dimension=None*)

Calculate weight according to joint limit.

changed()

Return False

This is used for CascadedCoords compatibility

Returns
False – always return False

Return type
bool

collision_avoidance_link_pair_from_link_list(*link_list*, *obstacles=None*)

compute_qp_common(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*)

compute_velocity(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*, *fast=True*, *sym_proj=False*, *solver='cvxopt'*, **args*, ***kwargs*)

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xy’, ‘yz’, ‘zx’, ‘xx’, ‘yy’, ‘zz’, True or False(None).

Returns

dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type

numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...           pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 2.0, 'y')
>>> c1.difference_position(c2)
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xx’, ‘yy’, ‘zz’, ‘xm’, ‘ym’, ‘zm’, ‘xy’, ‘yx’, ‘yz’, ‘zy’, ‘zx’, ‘xz’, True or False(None).

Returns
dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option [‘xm’, ‘ym’, ‘zm’], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

dissoc(*child*)

find_joint_angle_limit_weight_from_union_link_list(*union_link_list*)

find_link_path(*src_link*, *target_link*)
Find paths of src_link to target_link

Parameters

- **src_link** (*skrobot.model.link.Link*) – source link.
- **target_link** (*skrobot.model.link.Link*) – target link.

Returns
ret – If the links are connected, return Link list. Otherwise, return an empty list.

Return type
List[skrobot.model.link.Link]

find_link_route(*to*, *frm=None*)

fix_leg_to_coords(*fix_coords*, *leg='both'*, *mid=0.5*)

Fix robot's legs to a coords

In the Following codes, leged robot is assumed.

Parameters

- **fix_coords** (Coordinates) – target coordinate
- **leg** (*string*) – ['both', 'rleg', 'rleg', 'left', 'right']
- **mid** (*float*) – ratio of legs coord.

get_transform()

Return Transform object

Returns
transform – corrensponding Transform to this coordinates

Return type
skrobot.coordinates.base.Transform

ik_convergence_check(*dif_pos*, *dif_rot*, *rotation_axis*, *translation_axis*, *thre*, *rthre*, *centroid_thre=None*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis=None*, *update_mass_properties=True*)

check ik convergence.

Parameters

- **dif_pos** (*list of np.ndarray*) –
- **dif_rot** (*list of np.ndarray*) –
- **translation_axis** (*list of axis*) –
- **rotation_axis** (*list of axis*) – see _wrap_axis

init_pose()

inverse_kinematics(*target_coords*, *move_target=None*, *link_list=None*, ***kwargs*)

Solve inverse kinematics.

solve inverse kinematics, move move-target to target-coords look-at- target suppots t, nil, float-vector, coords, list of float-vector, list of coords link-list is set by default based on move-target -> root link link-list.

inverse_kinematics_args(*union_link_list=None*, *rotation_axis=None*, *translation_axis=None*, *additional_jacobi_dimension=0*, ***kwargs*)

inverse_kinematics_loop(*dif_pos*, *dif_rot*, *move_target*, *link_list=None*, *target_coords=None*, ***kwargs*)

move move_target using dif_pos and dif_rot.

Parameters
TODO –

Return type
TODO

inverse_kinematics_loop_for_look_at(*move_target*, *look_at*, *link_list*, *rotation_axis='z'*, *translation_axis=False*, *rthre=0.001*, ***kwargs*)

Solve look at inverse kinematics

Parameters
look_at (*list or np.ndarray or* Coordinates) –

inverse_kinematics_optimization(*target_coords*, *move_target=None*, *link_list=None*, *regularization_parameter=None*, *init_angle_vector=None*, *translation_axis=True*, *rotation_axis=True*, *stop=100*, *dt=0.005*, *inverse_kinematics_hook=[]*, *thre=0.001*, *rthre=0.017453292519943295*, **args*, ***kwargs*)

inverse_rotate_vector(*v*)

inverse_transform_vector(*v*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or* skrobot.coordinates.Coordinates) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

link_lists(*to*, *frm=None*)

Find link list from to link to frm link.

load_urdf(*urdf*)

load_urdf_file(*file_obj*)

look_at_hand(*coords*)

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.
- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

move_end_pos(*pos*, *wrt='local'*, **args*, ***kwargs*)

move_end_rot(*angle*, *axis*, *wrt='local'*, **args*, ***kwargs*)

move_joints(*union_vel*, *union_link_list=None*, *periodic_time=0.05*, *joint_args=None*, *move_joints_hook=None*, **args*, ***kwargs*)

move_joints_avoidance(*union_vel*, *union_link_list=None*, *link_list=None*, *n_joint_dimension=None*, *weight=None*, *null_space=None*, *avoid_nspace_gain=0.01*, *avoid_weight_gain=1.0*, *avoid_collision_distance=200*, *avoid_collision_null_gain=1.0*, *avoid_collision_joint_gain=1.0*, *collision_avoidance_link_pair=None*, *cog_gain=0.0*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis='z'*, *cog_null_space=False*, *additional_weight_list=None*, *additional_nspace_list=None*, *jacobi=None*, *obstacles=None*, **args*, ***kwargs*)

newcoords(*c*, *pos=None*, *check_validity=True*)

Update this coordinates.

This function records that this CascadedCoords has changed and recursively records the change to descendants of this CascadedCoords.

Parameters

- **c** (*skrobot.coordinates.Coordinates or numpy.ndarray*) – If pos is *None*, *c* means new Coordinates. If pos is given, *c* means rotation matrix.
- **pos** (*numpy.ndarray or None*) – new translation.
- **check_validity** (*bool*) – If this value is *True*, check whether an input rotation and an input translation are valid.

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (*numpy.ndarray*) – 3x3 rotation matrix.
- **wrt** (*str or skrobot.coordinates.Coordinates*) – reference coordinates.

parent_orientation(*v*, *wrt*)

parentcoords()

reset_joint_angle_limit_weight(*union_link_list*)

reset_manip_pose()

reset_pose()

rotate(*theta*, *axis*, *wrt='local'*)

Rotate this coordinate.

Rotate this coordinate relative to axis by theta radians with respect to wrt.

Parameters

- **theta** (*float*) – radian

- **axis** (*str or numpy.ndarray*) – 'x', 'y', 'z' or vector
- **wrt** (*str or Coordinates*) –

Return type
self

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (*numpy.ndarray*) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*matrix*, *wrt*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

self_collision_check()

Return collision link pair

Returns

- **is_collision** (*bool*) – True if a collision occurred between any pair of objects and False otherwise
- **names** (*set of 2-tuple*) – The set of pairwise collisions. Each tuple contains two names in alphabetical order indicating that the two corresponding objects are in collision.

transform(*c*, *wrt='local'*, *out=None*)

Transform this coordinates

Parameters

- **c** (skrobot.coordinates.Coordinates) – coordinates
- **wrt** (*str or* skrobot.coordinates.Coordinates) – transform this coordinates with respect to wrt. If wrt is 'local' or self, multiply c from the right. If wrt is 'parent' or self.parent, transform c with respect to parentcoords. (multiply c from the left.) If wrt is Coordinates, transform c with respect to c.
- **out** (*None or* skrobot.coordinates.Coordinates) – If the *out* is specified, set new coordinates to *out*. Note that if the *out* is given, these coordinates don't change.

Returns
self – return self

Return type
skrobot.coordinates.CascadedCoords

transform_vector(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

transformation(*c2*, *wrt='local'*)

translate(*vec*, *wrt='local'*)

Translate this coordinates.

Note that this function changes this coordinates self. So if you don't want to change this class, use copy_worldcoords()

Parameters

- **vec** (*list or numpy.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*str or Coordinates (optional)*) – translate with respect to wrt.

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.], dtype=float32)
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3], dtype=float32)
```

```
>>> c = Coordinates()
>>> c.copy_worldcoords().translate([0.1, 0.2, 0.3])
>>> c.translation
array([0., 0., 0.], dtype=float32)
```

```
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([ 0.3,  0.2, -0.1])
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3], 'world')
>>> c.translation
array([0.1, 0.2, 0.3])
```

update(*force=False*)

worldcoords()

Calculate rotation and position in the world.

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns

self.translation – translation of this coordinate

Return type

numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns

self.rotation – rotation matrix of this coordinate

Return type

numpy.ndarray

__eq__(*value*, /)
Return self==value.

__ne__(*value*, /)
Return self!=value.

__lt__(*value*, /)
Return self<value.

__le__(*value*, /)
Return self<=value.

__gt__(*value*, /)
Return self>value.

__ge__(*value*, /)
Return self>=value.

__mul__(*other_c*)
Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

__pow__(*exponent*)
Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

descendants

dimension
Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

dual_quaternion

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

interlocking_joint_pairs

Interlocking joint pairs.

pairs are [(joint0, joint1), ...] If users want to use interlocking joints, please overwrite this method.

joint_max_angles

joint_min_angles

larm

lleg

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

parent

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rarm

rleg

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

Return type
numpy.ndarray

`z_axis`

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray

3.2.2 Robot Model classes

You can create use robot model classes. Here is a example of robot models.

Fetch

skrobot.models.fetch.Fetch	Fetch Robot Model.

skrobot.models.fetch.Fetch

`class skrobot.models.fetch.Fetch(*args, **kwargs)`

Fetch Robot Model.

http://docs.fetchrobotics.com/robot_hardware.html

Methods

`T()`

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

(continues on next page)

(continued from previous page)

```
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

angle_vector(*av=None*, *return_av=None*)

Returns angle vector

If av is given, it updates angles of all joint. If given av violate min/max range, the value is modified.

assoc(*child*, *relative_coords='world'*, *force=False*, ***kwargs*)

Associate child coords to this coordinate system.

If *relative_coords* is *None* or 'world', the translation and rotation of childcoords in the world coordinate system do not change. If *relative_coords* is specified, childcoords is assoced at translation and rotation of *relative_coords*. By default, if child is already assoced to some other coords, raise an exception. But if *force* is *True*, you can overwrite the existing assoc relation.

Parameters

- **child** (CascadedCoords) – child coordinates.
- **relative_coords** (*None or* Coordinates *or str*) – child coordinates' relative coordinates.
- **force** (*bool*) – predicate for overwriting the existing assoc-relation

Returns

child – assoced child.

Return type

CascadedCoords

Examples

```
>>> from skrobot.coordinates import CascadedCoords
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords)
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
>>> child_coords.translation
array([0., 1., 0.])
```

None and 'world' have the same meaning.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
```

(continues on next page)

(continued from previous page)

```
>>> parent_coords.assoc(child_coords, relative_coords='world')
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
```

If *relative_coords* is 'local', *child* is associated at world translation and world rotation of *child* from this coordinate system.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='local')
>>> child_coords.worldpos()
array([2., 1., 0.])
>>> child_coords.translation
array([1., 1., 0.])
```

axis(*ax*)

calc_inverse_jacobian(*jacobi*, *manipulability_limit=0.1*, *manipulability_gain=0.001*, *weight=None*, **args*, ***kwargs*)

calc_inverse_kinematics_nspace_from_link_list(*link_list*, *avoid_nspace_gain=0.01*, *union_link_list=None*, *n_joint_dimension=None*, *null_space=None*, *additional_nspace_list=None*, *weight=None*)

calc_inverse_kinematics_weight_from_link_list(*link_list*, *avoid_weight_gain=1.0*, *union_link_list=None*, *n_joint_dimension=None*, *weight=None*, *additional_weight_list=[]*)

Calculate all weight from link list.

calc_jacobian_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

calc_jacobian_from_link_list(*move_target*, *link_list=None*, *transform_coords=None*, *rotation_axis=None*, *translation_axis=None*, *col_offset=0*, *dim=None*, *jacobian=None*, *additional_jacobi_dimension=0*, *n_joint_dimension=None*, **args*, ***kwargs*)

calc_joint_angle_from_min_max_table(*index*, *j*, *av*)

calc_joint_angle_speed(*union_vel*, *angle_speed=None*, *angle_speed_blending=0.5*, *jacobi=None*, *j_sharp=None*, *null_space=None*, **args*, ***kwargs*)

calc_joint_angle_speed_gain(*union_link_list*, *dav*, *periodic_time*)

calc_nspace_from_joint_limit(*avoid_nspace_gain*, *union_link_list*, *weight*)

Calculate null-space according to joint limit.

calc_target_axis_dimension(*rotation_axis*, *translation_axis*)

rotation-axis, translation-axis -> both list and atom OK.

calc_target_joint_dimension(*link_list*)

calc_union_link_list(*link_list*)

calc_vel_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

Calculate 0 velocity for keeping interlocking joint.

at the same joint angle.

calc_vel_from_pos(*dif_pos*, *translation_axis*, *p_limit=100.0*)

Calculate velocity from difference position

Parameters

- **dif_pos** (*np.ndarray*) – [m] order
- **translation_axis** (*str*) – see calc_dif_with_axis

Returns
vel_p

Return type
np.ndarray

calc_vel_from_rot(*dif_rot*, *rotation_axis*, *r_limit=0.5*)

calc_weight_from_joint_limit(*avoid_weight_gain*, *link_list*, *union_link_list*, *weight*, *n_joint_dimension=None*)

Calculate weight according to joint limit.

changed()

Return False

This is used for CascadedCoords compatibility

Returns
False – always return False

Return type
bool

collision_avoidance_link_pair_from_link_list(*link_list*, *obstacles=None*)

compute_qp_common(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*)

compute_velocity(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*, *fast=True*, *sym_proj=False*, *solver='cvxopt'*, **args*, ***kwargs*)

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xy’, ‘yz’, ‘zx’, ‘xx’, ‘yy’, ‘zz’, True or False(None).

Returns

dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type

numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...          pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 2.0, 'y')
>>> c1.difference_position(c2)
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xx’, ‘yy’, ‘zz’, ‘xm’, ‘ym’, ‘zm’, ‘xy’, ‘yx’, ‘yz’, ‘zy’, ‘zx’, ‘xz’, True or False(None).

Returns

dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type

numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option [‘xm’, ‘ym’, ‘zm’], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

dissoc(*child*)

find_joint_angle_limit_weight_from_union_link_list(*union_link_list*)

find_link_path(*src_link*, *target_link*)

Find paths of src_link to target_link

Parameters

- **src_link** (*skrobot.model.link.Link*) – source link.
- **target_link** (*skrobot.model.link.Link*) – target link.

Returns

ret – If the links are connected, return Link list. Otherwise, return an empty list.

Return type

List[skrobot.model.link.Link]

find_link_route(*to*, *frm=None*)

fix_leg_to_coords(*fix_coords*, *leg='both'*, *mid=0.5*)

Fix robot’s legs to a coords

In the Following codes, leged robot is assumed.

Parameters

- **fix_coords** (Coordinates) – target coordinate
- **leg** (*string*) – [‘both’, ‘rleg’, ‘rleg’, ‘left’, ‘right’]
- **mid** (*float*) – ratio of legs coord.

get_transform()

Return Transform object

Returns
transform – corrensponding Transform to this coordinates

Return type
skrobot.coordinates.base.Transform

ik_convergence_check(*dif_pos*, *dif_rot*, *rotation_axis*, *translation_axis*, *thre*, *rthre*, *centroid_thre=None*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis=None*, *update_mass_properties=True*)

check ik convergence.

Parameters

- **dif_pos** (*list of np.ndarray*) –
- **dif_rot** (*list of np.ndarray*) –
- **translation_axis** (*list of axis*) –
- **rotation_axis** (*list of axis*) – see _wrap_axis

init_pose()

inverse_kinematics(*target_coords*, *move_target=None*, *link_list=None*, ***kwargs*)

Solve inverse kinematics.

solve inverse kinematics, move move-target to target-coords look-at- target suppots t, nil, float-vector, coords, list of float-vector, list of coords link-list is set by default based on move-target -> root link link-list.

inverse_kinematics_args(*union_link_list=None*, *rotation_axis=None*, *translation_axis=None*, *additional_jacobi_dimension=0*, ***kwargs*)

inverse_kinematics_loop(*dif_pos*, *dif_rot*, *move_target*, *link_list=None*, *target_coords=None*, ***kwargs*)

move move_target using dif_pos and dif_rot.

Parameters
TODO –

Return type
TODO

inverse_kinematics_loop_for_look_at(*move_target*, *look_at*, *link_list*, *rotation_axis='z'*, *translation_axis=False*, *rthre=0.001*, ***kwargs*)

Solve look at inverse kinematics

Parameters
look_at (*list or np.ndarray or* Coordinates) –

inverse_kinematics_optimization(*target_coords*, *move_target=None*, *link_list=None*, *regularization_parameter=None*, *init_angle_vector=None*, *translation_axis=True*, *rotation_axis=True*, *stop=100*, *dt=0.005*, *inverse_kinematics_hook=[]*, *thre=0.001*, *rthre=0.017453292519943295*, **args*, ***kwargs*)

inverse_rotate_vector(*v*)

inverse_transform_vector(*v*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or skrobot.coordinates.Coordinates*) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

link_lists(*to*, *frm=None*)

Find link list from to link to frm link.

load_urdf(*urdf*)

load_urdf_file(*file_obj*)

look_at_hand(*coords*)

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.
- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

move_end_pos(*pos*, *wrt='local'*, **args*, ***kwargs*)

move_end_rot(*angle*, *axis*, *wrt='local'*, **args*, ***kwargs*)

move_joints(*union_vel*, *union_link_list=None*, *periodic_time=0.05*, *joint_args=None*, *move_joints_hook=None*, **args*, ***kwargs*)

move_joints_avoidance(*union_vel*, *union_link_list=None*, *link_list=None*, *n_joint_dimension=None*, *weight=None*, *null_space=None*, *avoid_nspace_gain=0.01*, *avoid_weight_gain=1.0*, *avoid_collision_distance=200*, *avoid_collision_null_gain=1.0*, *avoid_collision_joint_gain=1.0*, *collision_avoidance_link_pair=None*, *cog_gain=0.0*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis='z'*, *cog_null_space=False*, *additional_weight_list=None*, *additional_nspace_list=None*, *jacobi=None*, *obstacles=None*, **args*, ***kwargs*)

newcoords(*c*, *pos=None*, *check_validity=True*)

Update this coordinates.

This function records that this CascadedCoords has changed and recursively records the change to descendants of this CascadedCoords.

Parameters

- **c** (*skrobot.coordinates.Coordinates or numpy.ndarray*) – If pos is *None*, *c* means new Coordinates. If pos is given, *c* means rotation matrix.
- **pos** (*numpy.ndarray or None*) – new translation.
- **check_validity** (*bool*) – If this value is *True*, check whether an input rotation and an input translation are valid.

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (*numpy.ndarray*) – 3x3 rotation matrix.
- **wrt** (*str or skrobot.coordinates.Coordinates*) – reference coordinates.

parent_orientation(*v*, *wrt*)

parentcoords()

reset_joint_angle_limit_weight(*union_link_list*)

reset_manip_pose()

reset_pose()

rotate(*theta*, *axis*, *wrt='local'*)

Rotate this coordinate.

Rotate this coordinate relative to axis by theta radians with respect to wrt.

Parameters

- **theta** (*float*) – radian
- **axis** (*str or numpy.ndarray*) – 'x', 'y', 'z' or vector
- **wrt** (*str or Coordinates*) –

Return type
self

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (*numpy.ndarray*) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*matrix*, *wrt*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

self_collision_check()

Return collision link pair

Returns

- **is_collision** (*bool*) – True if a collision occurred between any pair of objects and False otherwise
- **names** (*set of 2-tuple*) – The set of pairwise collisions. Each tuple contains two names in alphabetical order indicating that the two corresponding objects are in collision.

transform(*c*, *wrt='local'*, *out=None*)

Transform this coordinates

Parameters

- **c** (skrobot.coordinates.Coordinates) – coordinates
- **wrt** (*str or* skrobot.coordinates.Coordinates) – transform this coordinates with respect to wrt. If wrt is 'local' or self, multiply c from the right. If wrt is 'parent' or self.parent, transform c with respect to parentcoords. (multiply c from the left.) If wrt is Coordinates, transform c with respect to c.
- **out** (*None or* skrobot.coordinates.Coordinates) – If the *out* is specified, set new coordinates to *out*. Note that if the *out* is given, these coordinates don't change.

Returns
self – return self

Return type
skrobot.coordinates.CascadedCoords

transform_vector(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

transformation(*c2*, *wrt='local'*)

translate(*vec*, *wrt='local'*)

Translate this coordinates.

Note that this function changes this coordinates self. So if you don't want to change this class, use copy_worldcoords()

Parameters

- **vec** (*list or numpy.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*str or* Coordinates *(optional)*) – translate with respect to wrt.

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.], dtype=float32)
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3], dtype=float32)
```

```
>>> c = Coordinates()
>>> c.copy_worldcoords().translate([0.1, 0.2, 0.3])
>>> c.translation
array([0., 0., 0.], dtype=float32)
```

```
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([ 0.3,  0.2, -0.1])
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3], 'world')
>>> c.translation
array([0.1, 0.2, 0.3])
```

update(*force=False*)

worldcoords()

Calculate rotation and position in the world.

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns
self.translation – translation of this coordinate

Return type
numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns
self.rotation – rotation matrix of this coordinate

Return type
numpy.ndarray

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, /)

Return self>=value.

__mul__(*other_c*)

Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

__pow__(*exponent*)

Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

default_urdf_path

descendants

dimension

Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

dual_quaternion

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

interlocking_joint_pairs

Interlocking joint pairs.

pairs are [(joint0, joint1), ...] If users want to use interlocking joints, please overwrite this method.

joint_max_angles

joint_min_angles

larm

lleg

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

parent

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rarm

rleg

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

Return type
numpy.ndarray

z_axis

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray

Kuka

skrobot.models.kuka.Kuka	Kuka Robot Model.

skrobot.models.kuka.Kuka

class skrobot.models.kuka.**Kuka**(**args*, ***kwargs*)

Kuka Robot Model.

Methods

T()

Return 4x4 homogeneous transformation matrix.

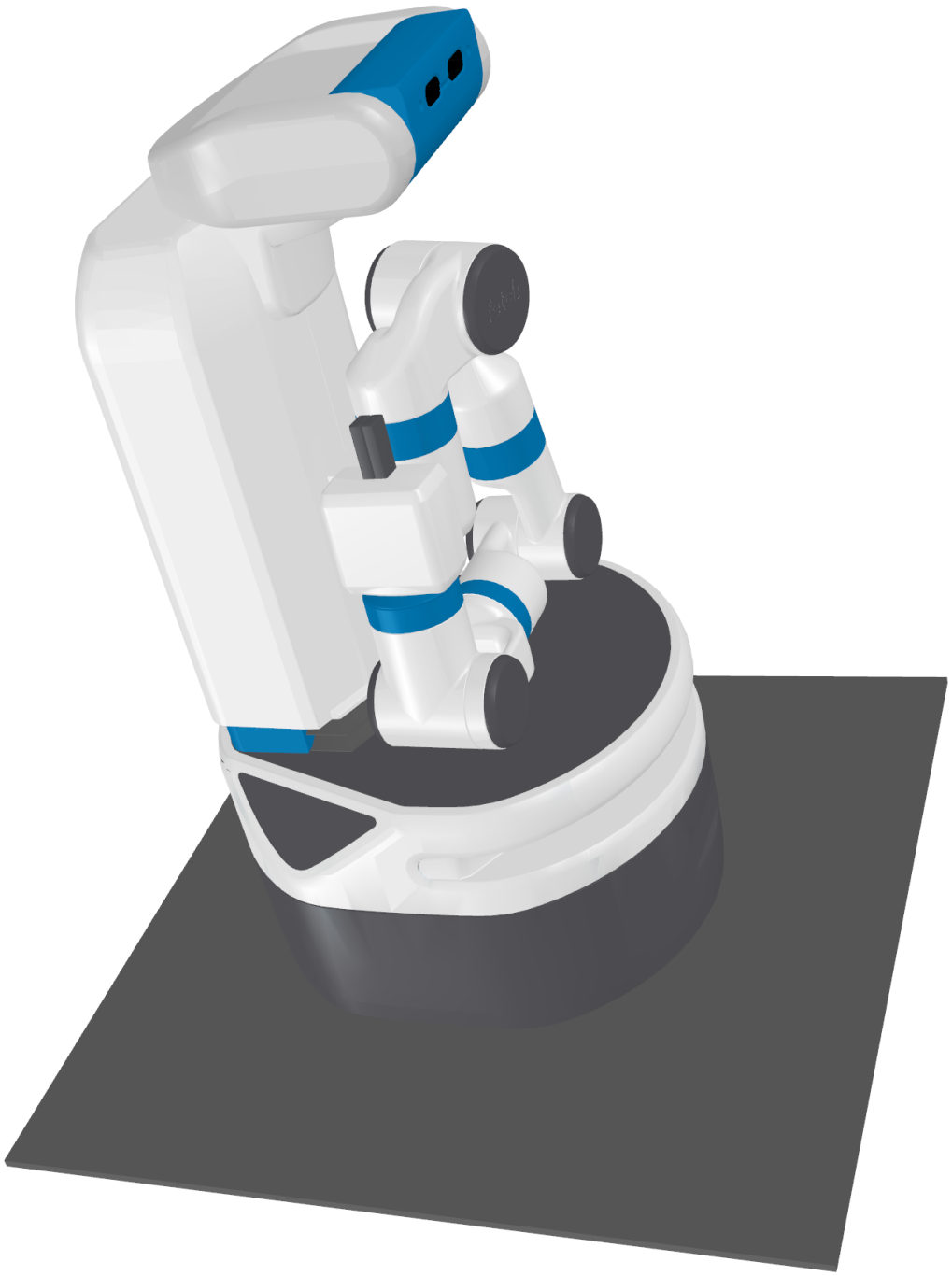
Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
```

(continues on next page)



(continued from previous page)

```
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

angle_vector(*av=None*, *return_av=None*)

Returns angle vector

If av is given, it updates angles of all joint. If given av violate min/max range, the value is modified.

assoc(*child*, *relative_coords='world'*, *force=False*, ***kwargs*)

Associate child coords to this coordinate system.

If *relative_coords* is *None* or 'world', the translation and rotation of childcoords in the world coordinate system do not change. If *relative_coords* is specified, childcoords is assoced at translation and rotation of *relative_coords*. By default, if child is already assoced to some other coords, raise an exception. But if *force* is *True*, you can overwrite the existing assoc relation.

Parameters

- **child** (CascadedCoords) – child coordinates.
- **relative_coords** (*None or Coordinates or str*) – child coordinates' relative coordinates.
- **force** (*bool*) – predicate for overwriting the existing assoc-relation

Returns

child – assoced child.

Return type

CascadedCoords

Examples

```
>>> from skrobot.coordinates import CascadedCoords
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords)
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
>>> child_coords.translation
array([0., 1., 0.])
```

None and 'world' have the same meaning.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='world')
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
```

If *relative_coords* is 'local', *child* is associated at world translation and world rotation of *child* from this coordinate system.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='local')
>>> child_coords.worldpos()
array([2., 1., 0.])
>>> child_coords.translation
array([1., 1., 0.])
```

axis(*ax*)

calc_inverse_jacobian(*jacobi*, *manipulability_limit=0.1*, *manipulability_gain=0.001*, *weight=None*, **args*, ***kwargs*)

calc_inverse_kinematics_nspace_from_link_list(*link_list*, *avoid_nspace_gain=0.01*, *union_link_list=None*, *n_joint_dimension=None*, *null_space=None*, *additional_nspace_list=None*, *weight=None*)

calc_inverse_kinematics_weight_from_link_list(*link_list*, *avoid_weight_gain=1.0*, *union_link_list=None*, *n_joint_dimension=None*, *weight=None*, *additional_weight_list=[]*)

Calculate all weight from link list.

calc_jacobian_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

calc_jacobian_from_link_list(*move_target*, *link_list=None*, *transform_coords=None*, *rotation_axis=None*, *translation_axis=None*, *col_offset=0*, *dim=None*, *jacobian=None*, *additional_jacobi_dimension=0*, *n_joint_dimension=None*, **args*, ***kwargs*)

calc_joint_angle_from_min_max_table(*index*, *j*, *av*)

calc_joint_angle_speed(*union_vel*, *angle_speed=None*, *angle_speed_blending=0.5*, *jacobi=None*, *j_sharp=None*, *null_space=None*, **args*, ***kwargs*)

calc_joint_angle_speed_gain(*union_link_list*, *dav*, *periodic_time*)

calc_nspace_from_joint_limit(*avoid_nspace_gain*, *union_link_list*, *weight*)

Calculate null-space according to joint limit.

calc_target_axis_dimension(*rotation_axis*, *translation_axis*)

rotation-axis, translation-axis -> both list and atom OK.

calc_target_joint_dimension(*link_list*)

calc_union_link_list(*link_list*)

calc_vel_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

Calculate 0 velocity for keeping interlocking joint.

at the same joint angle.

calc_vel_from_pos(*dif_pos*, *translation_axis*, *p_limit=100.0*)

Calculate velocity from difference position

Parameters

- **dif_pos** (*np.ndarray*) – [m] order

- **translation_axis** (*str*) – see calc_dif_with_axis

Returns
vel_p

Return type
np.ndarray

calc_vel_from_rot(*dif_rot*, *rotation_axis*, *r_limit=0.5*)

calc_weight_from_joint_limit(*avoid_weight_gain*, *link_list*, *union_link_list*, *weight*, *n_joint_dimension=None*)

Calculate weight according to joint limit.

changed()

Return False

This is used for CascadedCoords compatibility

Returns
False – always return False

Return type
bool

close_hand(*av=None*)

collision_avoidance_link_pair_from_link_list(*link_list*, *obstacles=None*)

compute_qp_common(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*)

compute_velocity(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*, *fast=True*, *sym_proj=False*, *solver='cvxopt'*, **args*, ***kwargs*)

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xy’, ‘yz’, ‘zx’, ‘xx’, ‘yy’, ‘zz’, True or False(None).

Returns
dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...           pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 2.0, 'y')
>>> c1.difference_position(c2)
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xx’, ‘yy’, ‘zz’, ‘xm’, ‘ym’, ‘zm’, ‘xy’, ‘yx’, ‘yz’, ‘zy’, ‘zx’, ‘xz’, True or False(None).

Returns
dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option ['xm', 'ym', 'zm'], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

dissoc(*child*)

find_joint_angle_limit_weight_from_union_link_list(*union_link_list*)

find_link_path(*src_link*, *target_link*)

Find paths of src_link to target_link

Parameters

- **src_link** (*skrobot.model.link.Link*) – source link.
- **target_link** (*skrobot.model.link.Link*) – target link.

Returns

ret – If the links are connected, return Link list. Otherwise, return an empty list.

Return type

List[skrobot.model.link.Link]

find_link_route(*to*, *frm=None*)

fix_leg_to_coords(*fix_coords*, *leg='both'*, *mid=0.5*)

Fix robot's legs to a coords

In the Following codes, leged robot is assumed.

Parameters

- **fix_coords** (Coordinates) – target coordinate
- **leg** (*string*) – ['both', 'rleg', 'rleg', 'left', 'right']
- **mid** (*float*) – ratio of legs coord.

get_transform()

Return Transform object

Returns

transform – corrensponding Transform to this coordinates

Return type

skrobot.coordinates.base.Transform

ik_convergence_check(*dif_pos*, *dif_rot*, *rotation_axis*, *translation_axis*, *thre*, *rthre*, *centroid_thre=None*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis=None*, *update_mass_properties=True*)

check ik convergence.

Parameters

- **dif_pos** (*list of np.ndarray*) –
- **dif_rot** (*list of np.ndarray*) –
- **translation_axis** (*list of axis*) –
- **rotation_axis** (*list of axis*) – see _wrap_axis

init_pose()

inverse_kinematics(*target_coords*, *move_target=None*, *link_list=None*, ***kwargs*)

Solve inverse kinematics.

solve inverse kinematics, move move-target to target-coords look-at- target suppots t, nil, float-vector, coords, list of float-vector, list of coords link-list is set by default based on move-target -> root link link-list.

inverse_kinematics_args(*union_link_list=None*, *rotation_axis=None*, *translation_axis=None*, *additional_jacobi_dimension=0*, ***kwargs*)

inverse_kinematics_loop(*dif_pos*, *dif_rot*, *move_target*, *link_list=None*, *target_coords=None*, ***kwargs*)

move move_target using dif_pos and dif_rot.

Parameters
TODO –

Return type
TODO

inverse_kinematics_loop_for_look_at(*move_target*, *look_at*, *link_list*, *rotation_axis='z'*, *translation_axis=False*, *rthre=0.001*, ***kwargs*)

Solve look at inverse kinematics

Parameters
look_at (*list or np.ndarray or Coordinates*) –

inverse_kinematics_optimization(*target_coords*, *move_target=None*, *link_list=None*, *regularization_parameter=None*, *init_angle_vector=None*, *translation_axis=True*, *rotation_axis=True*, *stop=100*, *dt=0.005*, *inverse_kinematics_hook=[]*, *thre=0.001*, *rthre=0.017453292519943295*, **args*, ***kwargs*)

inverse_rotate_vector(*v*)

inverse_transform_vector(*v*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or skrobot.coordinates.Coordinates*) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

link_lists(*to*, *frm=None*)

Find link list from to link to frm link.

load_urdf(*urdf*)

load_urdf_file(*file_obj*)

look_at_hand(*coords*)

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.
- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

move_end_pos(*pos*, *wrt='local'*, **args*, ***kwargs*)

move_end_rot(*angle*, *axis*, *wrt='local'*, **args*, ***kwargs*)

move_joints(*union_vel*, *union_link_list=None*, *periodic_time=0.05*, *joint_args=None*, *move_joints_hook=None*, **args*, ***kwargs*)

move_joints_avoidance(*union_vel*, *union_link_list=None*, *link_list=None*, *n_joint_dimension=None*, *weight=None*, *null_space=None*, *avoid_nspace_gain=0.01*, *avoid_weight_gain=1.0*, *avoid_collision_distance=200*, *avoid_collision_null_gain=1.0*, *avoid_collision_joint_gain=1.0*, *collision_avoidance_link_pair=None*, *cog_gain=0.0*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis='z'*, *cog_null_space=False*, *additional_weight_list=None*, *additional_nspace_list=None*, *jacobi=None*, *obstacles=None*, **args*, ***kwargs*)

newcoords(*c*, *pos=None*, *check_validity=True*)

Update this coordinates.

This function records that this CascadedCoords has changed and recursively records the change to descendants of this CascadedCoords.

Parameters

- **c** (skrobot.coordinates.Coordinates *or* numpy.ndarray) – If pos is *None*, *c* means new Coordinates. If pos is given, *c* means rotation matrix.
- **pos** (numpy.ndarray *or None*) – new translation.

- **check_validity** (*bool*) – If this value is *True*, check whether an input rotation and an input translation are valid.

open_hand(*default_angle=0.17453292519943295*, *av=None*)

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (*numpy.ndarray*) – 3x3 rotation matrix.
- **wrt** (*str or skrobot.coordinates.Coordinates*) – reference coordinates.

parent_orientation(*v*, *wrt*)

parentcoords()

reset_joint_angle_limit_weight(*union_link_list*)

reset_manip_pose()

reset_pose()

rotate(*theta*, *axis*, *wrt='local'*)

Rotate this coordinate.

Rotate this coordinate relative to axis by theta radians with respect to wrt.

Parameters

- **theta** (*float*) – radian
- **axis** (*str or numpy.ndarray*) – 'x', 'y', 'z' or vector
- **wrt** (*str or Coordinates*) –

Return type
self

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (*numpy.ndarray*) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*matrix*, *wrt*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

self_collision_check()

Return collision link pair

Returns

- **is_collision** (*bool*) – True if a collision occurred between any pair of objects and False otherwise
- **names** (*set of 2-tuple*) – The set of pairwise collisions. Each tuple contains two names in alphabetical order indicating that the two corresponding objects are in collision.

transform(*c*, *wrt='local'*, *out=None*)

Transform this coordinates

Parameters

- **c** (*skrobot.coordinates.Coordinates*) – coordinates
- **wrt** (*str or skrobot.coordinates.Coordinates*) – transform this coordinates with respect to wrt. If wrt is 'local' or self, multiply c from the right. If wrt is 'parent' or self.parent, transform c with respect to parentcoords. (multiply c from the left.) If wrt is Coordinates, transform c with respect to c.
- **out** (*None or skrobot.coordinates.Coordinates*) – If the *out* is specified, set new coordinates to *out*. Note that if the *out* is given, these coordinates don't change.

Returns
self – return self

Return type
skrobot.coordinates.CascadedCoords

`transform_vector`(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

`transformation`(*c2*, *wrt='local'*)

`translate`(*vec*, *wrt='local'*)

Translate this coordinates.

Note that this function changes this coordinates self. So if you don't want to change this class, use copy_worldcoords()

Parameters

- **vec** (*list or numpy.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*str or Coordinates (optional)*) – translate with respect to wrt.

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.], dtype=float32)
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3], dtype=float32)
```

```
>>> c = Coordinates()
>>> c.copy_worldcoords().translate([0.1, 0.2, 0.3])
>>> c.translation
array([0., 0., 0.], dtype=float32)
```

```
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([ 0.3,  0.2, -0.1])
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3], 'world')
```

(continues on next page)

(continued from previous page)

```
>>> c.translation
array([0.1, 0.2, 0.3])
```

update(*force=False*)

worldcoords()

Calculate rotation and position in the world.

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns
self.translation – translation of this coordinate

Return type
numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns
self.rotation – rotation matrix of this coordinate

Return type
numpy.ndarray

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)

Return self<=value.

__gt__(*value*, */*)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

__mul__(*other_c*)

Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

`__pow__`(*exponent*)

Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

`default_urdf_path`

`descendants`

`dimension`

Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

`dual_quaternion`

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

`interlocking_joint_pairs`

Interlocking joint pairs.

pairs are [(joint0, joint1), ...] If users want to use interlocking joints, please overwrite this method.

`joint_max_angles`

`joint_min_angles`

`larm`

`lleg`

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

parent

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rarm

rleg

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
```

(continues on next page)

(continued from previous page)

```
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

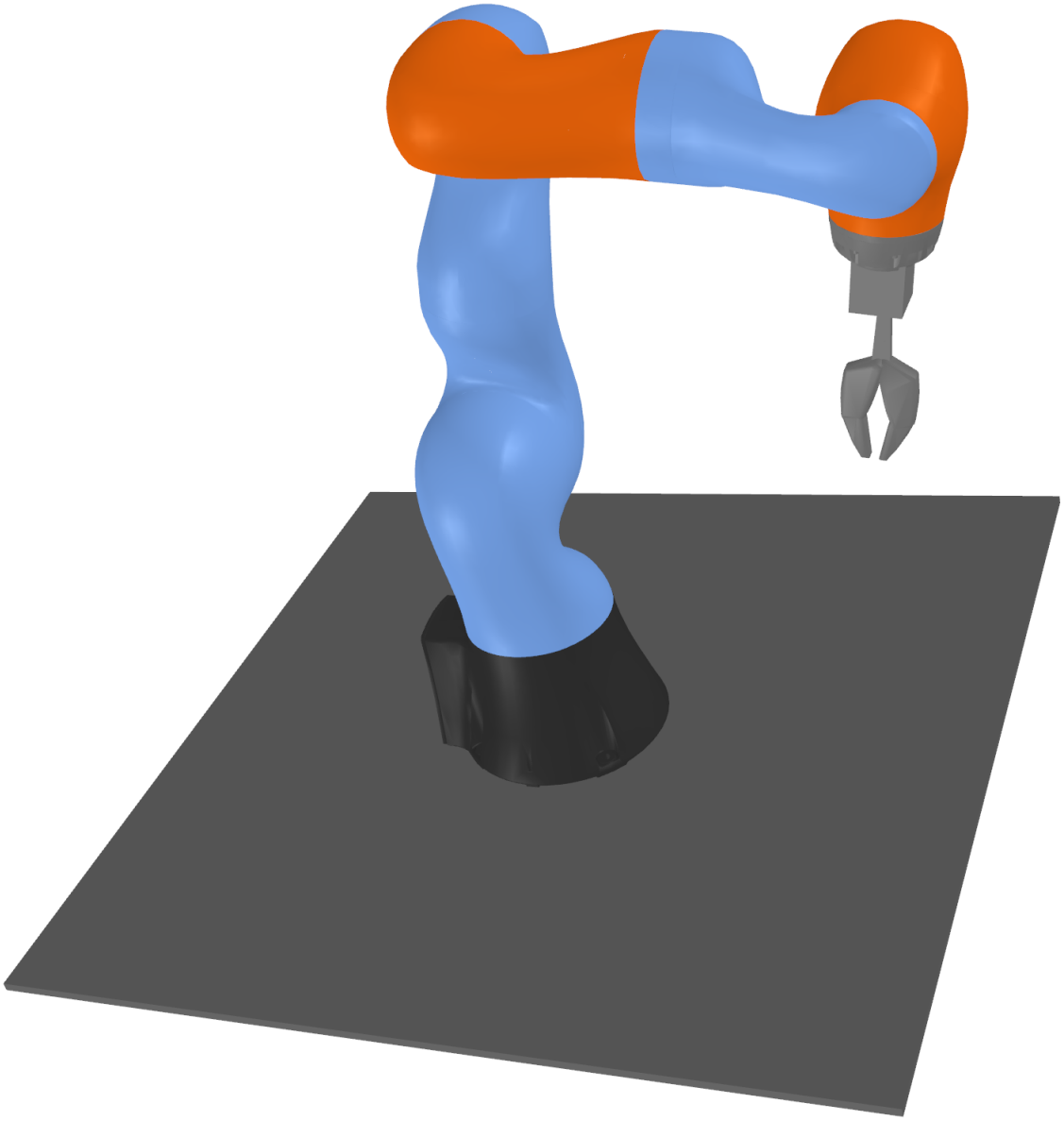
Return type
numpy.ndarray

z_axis

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray



PR2

skrobot.models.pr2.PR2	PR2 Robot Model.

skrobot.models.pr2.PR2

class skrobot.models.pr2.**PR2**(*use_tight_joint_limit=True*)

PR2 Robot Model.

Methods

T()

Return 4x4 homogeneous transformation matrix.

Returns
matrix – homogeneous transformation matrix shape of (4, 4)

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

angle_vector(*av=None*, *return_av=None*)

Returns angle vector

If av is given, it updates angles of all joint. If given av violate min/max range, the value is modified.

assoc(*child*, *relative_coords='world'*, *force=False*, ***kwargs*)

Associate child coords to this coordinate system.

If *relative_coords* is *None* or 'world', the translation and rotation of childcoords in the world coordinate system do not change. If *relative_coords* is specified, childcoords is assoced at translation and rotation of

relative_coords. By default, if child is already assoced to some other coords, raise an exception. But if *force* is *True*, you can overwrite the existing assoc relation.

Parameters

- **child** (CascadedCoords) – child coordinates.
- **relative_coords** (*None or Coordinates or str*) – child coordinates' relative coordinates.
- **force** (*bool*) – predicate for overwriting the existing assoc-relation

Returns
child – assoced child.

Return type
CascadedCoords

Examples

```
>>> from skrobot.coordinates import CascadedCoords
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords)
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
>>> child_coords.translation
array([0., 1., 0.])
```

None and 'world' have the same meaning.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='world')
#<CascadedCoords 0x7f1d30e29510 1.000 1.000 0.000 / 0.0 -0.0 0.0>
>>> child_coords.worldpos()
array([1., 1., 0.])
```

If *relative_coords* is 'local', *child* is associated at world translation and world rotation of *child* from this coordinate system.

```
>>> parent_coords = CascadedCoords(pos=[1, 0, 0])
>>> child_coords = CascadedCoords(pos=[1, 1, 0])
>>> parent_coords.assoc(child_coords, relative_coords='local')
>>> child_coords.worldpos()
array([2., 1., 0.])
>>> child_coords.translation
array([1., 1., 0.])
```

axis(*ax*)

calc_inverse_jacobian(*jacobi*, *manipulability_limit=0.1*, *manipulability_gain=0.001*, *weight=None*, **args*, ***kwargs*)

calc_inverse_kinematics_nspace_from_link_list(*link_list*, *avoid_nspace_gain=0.01*, *union_link_list=None*, *n_joint_dimension=None*, *null_space=None*, *additional_nspace_list=None*, *weight=None*)

calc_inverse_kinematics_weight_from_link_list(*link_list*, *avoid_weight_gain=1.0*, *union_link_list=None*, *n_joint_dimension=None*, *weight=None*, *additional_weight_list=[]*)

Calculate all weight from link list.

calc_jacobian_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

calc_jacobian_from_link_list(*move_target*, *link_list=None*, *transform_coords=None*, *rotation_axis=None*, *translation_axis=None*, *col_offset=0*, *dim=None*, *jacobian=None*, *additional_jacobi_dimension=0*, *n_joint_dimension=None*, **args*, ***kwargs*)

calc_joint_angle_from_min_max_table(*index*, *j*, *av*)

calc_joint_angle_speed(*union_vel*, *angle_speed=None*, *angle_speed_blending=0.5*, *jacobi=None*, *j_sharp=None*, *null_space=None*, **args*, ***kwargs*)

calc_joint_angle_speed_gain(*union_link_list*, *dav*, *periodic_time*)

calc_nspace_from_joint_limit(*avoid_nspace_gain*, *union_link_list*, *weight*)

Calculate null-space according to joint limit.

calc_target_axis_dimension(*rotation_axis*, *translation_axis*)

rotation-axis, translation-axis -> both list and atom OK.

calc_target_joint_dimension(*link_list*)

calc_union_link_list(*link_list*)

calc_vel_for_interlocking_joints(*link_list*, *interlocking_joint_pairs=None*)

Calculate 0 velocity for keeping interlocking joint.

at the same joint angle.

calc_vel_from_pos(*dif_pos*, *translation_axis*, *p_limit=100.0*)

Calculate velocity from difference position

Parameters

- **dif_pos** (*np.ndarray*) – [m] order
- **translation_axis** (*str*) – see calc_dif_with_axis

Returns
vel_p

Return type
np.ndarray

calc_vel_from_rot(*dif_rot*, *rotation_axis*, *r_limit=0.5*)

calc_weight_from_joint_limit(*avoid_weight_gain*, *link_list*, *union_link_list*, *weight*, *n_joint_dimension=None*)

Calculate weight according to joint limit.

changed()

Return False

This is used for CascadedCoords compatibility

Returns
False – always return False

Return type
bool

collision_avoidance_link_pair_from_link_list(*link_list*, *obstacles=None*)

compute_qp_common(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*)

compute_velocity(*target_coords*, *move_target*, *dt*, *link_list=None*, *gain=0.85*, *weight=1.0*, *translation_axis=True*, *rotation_axis=True*, *dof_limit_gain=0.5*, *fast=True*, *sym_proj=False*, *solver='cvxopt'*, **args*, ***kwargs*)

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xy’, ‘yz’, ‘zx’, ‘xx’, ‘yy’, ‘zz’, True or False(None).

Returns
dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...           pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...           pi / 2.0, 'y')
>>> c1.difference_position(c2)
```

(continues on next page)

(continued from previous page)

```
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take ‘x’, ‘y’, ‘z’, ‘xx’, ‘yy’, ‘zz’, ‘xm’, ‘ym’, ‘zm’, ‘xy’, ‘yx’, ‘yz’, ‘zy’, ‘zx’, ‘xz’, True or False(None).

Returns

dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type

numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option [‘xm’, ‘ym’, ‘zm’], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

dissoc(*child*)

find_joint_angle_limit_weight_from_union_link_list(*union_link_list*)

find_link_path(*src_link*, *target_link*)

Find paths of src_link to target_link

Parameters

- **src_link** (*skrobot.model.link.Link*) – source link.
- **target_link** (*skrobot.model.link.Link*) – target link.

Returns

ret – If the links are connected, return Link list. Otherwise, return an empty list.

Return type

List[skrobot.model.link.Link]

find_link_route(*to*, *frm=None*)

fix_leg_to_coords(*fix_coords*, *leg='both'*, *mid=0.5*)

Fix robot's legs to a coords

In the Following codes, leged robot is assumed.

Parameters

- **fix_coords** (Coordinates) – target coordinate
- **leg** (*string*) – ['both', 'rleg', 'rleg', 'left', 'right']
- **mid** (*float*) – ratio of legs coord.

get_transform()

Return Transform object

Returns

transform – correnssponding Transform to this coordinates

Return type

skrobot.coordinates.base.Transform

gripper_distance(*dist=None*, *arm='arms'*)

Change gripper angle function

Parameters

- **dist** (*None or float*) – gripper distance. If dist is None, return gripper distance. If flaot value is given, change joint angle.
- **arm** (*str*) – Specify target arm. You can specify 'larm', 'rarm', 'arms'.

Returns

dist – Result of gripper distance in meter.

Return type

float

ik_convergence_check(*dif_pos*, *dif_rot*, *rotation_axis*, *translation_axis*, *thre*, *rthre*, *centroid_thre=None*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis=None*, *update_mass_properties=True*)

check ik convergence.

Parameters

- **dif_pos** (*list of np.ndarray*) –
- **dif_rot** (*list of np.ndarray*) –
- **translation_axis** (*list of axis*) –
- **rotation_axis** (*list of axis*) – see _wrap_axis

init_pose()

inverse_kinematics(*target_coords*, *move_target=None*, *link_list=None*, ***kwargs*)

Solve inverse kinematics.

solve inverse kinematics, move move-target to target-coords look-at- target suppots t, nil, float-vector, coords, list of float-vector, list of coords link-list is set by default based on move-target -> root link link-list.

inverse_kinematics_args(*union_link_list=None*, *rotation_axis=None*, *translation_axis=None*, *additional_jacobi_dimension=0*, ***kwargs*)

inverse_kinematics_loop(*dif_pos*, *dif_rot*, *move_target*, *link_list=None*, *target_coords=None*, ***kwargs*)

move move_target using dif_pos and dif_rot.

Parameters
TODO –

Return type
TODO

inverse_kinematics_loop_for_look_at(*move_target*, *look_at*, *link_list*, *rotation_axis='z'*, *translation_axis=False*, *rthre=0.001*, ***kwargs*)

Solve look at inverse kinematics

Parameters
look_at (*list or np.ndarray or Coordinates*) –

inverse_kinematics_optimization(*target_coords*, *move_target=None*, *link_list=None*, *regularization_parameter=None*, *init_angle_vector=None*, *translation_axis=True*, *rotation_axis=True*, *stop=100*, *dt=0.005*, *inverse_kinematics_hook=[]*, *thre=0.001*, *rthre=0.017453292519943295*, **args*, ***kwargs*)

inverse_rotate_vector(*v*)

inverse_transform_vector(*v*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or* skrobot.coordinates.Coordinates) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

link_lists(*to*, *frm=None*)

Find link list from to link to frm link.

load_urdf(*urdf*)

load_urdf_file(*file_obj*)

look_at_hand(*coords*)

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.
- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

move_end_pos(*pos*, *wrt='local'*, **args*, ***kwargs*)

move_end_rot(*angle*, *axis*, *wrt='local'*, **args*, ***kwargs*)

move_joints(*union_vel*, *union_link_list=None*, *periodic_time=0.05*, *joint_args=None*, *move_joints_hook=None*, **args*, ***kwargs*)

move_joints_avoidance(*union_vel*, *union_link_list=None*, *link_list=None*, *n_joint_dimension=None*, *weight=None*, *null_space=None*, *avoid_nspace_gain=0.01*, *avoid_weight_gain=1.0*, *avoid_collision_distance=200*, *avoid_collision_null_gain=1.0*, *avoid_collision_joint_gain=1.0*, *collision_avoidance_link_pair=None*, *cog_gain=0.0*, *target_centroid_pos=None*, *centroid_offset_func=None*, *cog_translation_axis='z'*, *cog_null_space=False*, *additional_weight_list=None*, *additional_nspace_list=None*, *jacobi=None*, *obstacles=None*, **args*, ***kwargs*)

newcoords(*c*, *pos=None*, *check_validity=True*)

Update this coordinates.

This function records that this CascadedCoords has changed and recursively records the change to descendants of this CascadedCoords.

Parameters

- **c** (skrobot.coordinates.Coordinates *or* numpy.ndarray) – If pos is *None*, *c* means new Coordinates. If pos is given, *c* means rotation matrix.
- **pos** (numpy.ndarray *or* None) – new translation.
- **check_validity** (bool) – If this value is *True*, check whether an input rotation and an input translation are valid.

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (numpy.ndarray) – 3x3 rotation matrix.
- **wrt** (str *or* skrobot.coordinates.Coordinates) – reference coordinates.

parent_orientation(*v*, *wrt*)

parentcoords()

reset_joint_angle_limit_weight(*union_link_list*)

reset_manip_pose()

reset_pose()

rotate(*theta*, *axis*, *wrt='local'*)

Rotate this coordinate.

Rotate this coordinate relative to axis by theta radians with respect to wrt.

Parameters

- **theta** (float) – radian
- **axis** (str *or* numpy.ndarray) – 'x', 'y', 'z' or vector
- **wrt** (str *or* Coordinates) –

Return type
self

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (numpy.ndarray) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*matrix*, *wrt*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

self_collision_check()

Return collision link pair

Returns

- **is_collision** (*bool*) – True if a collision occurred between any pair of objects and False otherwise
- **names** (*set of 2-tuple*) – The set of pairwise collisions. Each tuple contains two names in alphabetical order indicating that the two corresponding objects are in collision.

transform(*c*, *wrt='local'*, *out=None*)

Transform this coordinates

Parameters

- **c** (skrobot.coordinates.Coordinates) – coordinates
- **wrt** (*str or* skrobot.coordinates.Coordinates) – transform this coordinates with respect to wrt. If wrt is 'local' or self, multiply c from the right. If wrt is 'parent' or self.parent, transform c with respect to parentcoords. (multiply c from the left.) If wrt is Coordinates, transform c with respect to c.
- **out** (*None or* skrobot.coordinates.Coordinates) – If the *out* is specified, set new coordinates to *out*. Note that if the *out* is given, these coordinates don't change.

Returns
self – return self

Return type
skrobot.coordinates.CascadedCoords

transform_vector(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

transformation(*c2*, *wrt='local'*)

translate(*vec*, *wrt='local'*)

Translate this coordinates.

Note that this function changes this coordinates self. So if you don't want to change this class, use copy_worldcoords()

Parameters

- **vec** (*list or numpy.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*str or* Coordinates *(optional)*) – translate with respect to wrt.

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.], dtype=float32)
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3], dtype=float32)
```

```
>>> c = Coordinates()
>>> c.copy_worldcoords().translate([0.1, 0.2, 0.3])
>>> c.translation
array([0., 0., 0.], dtype=float32)
```

```
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([ 0.3,  0.2, -0.1])
>>> c = Coordinates().rotate(np.pi / 2.0, 'y')
>>> c.translate([0.1, 0.2, 0.3], 'world')
>>> c.translation
array([0.1, 0.2, 0.3])
```

update(*force=False*)

worldcoords()

Calculate rotation and position in the world.

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns
self.translation – translation of this coordinate

Return type
numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns
self.rotation – rotation matrix of this coordinate

Return type
numpy.ndarray

__eq__(*value*, /)

Return self==value.

__ne__(*value*, /)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, /)

Return self>=value.

__mul__(*other_c*)

Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

__pow__(*exponent*)

Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

default_urdf_path

descendants

dimension

Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

dual_quaternion

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

head

interlocking_joint_pairs

Interlocking joint pairs.

pairs are [(joint0, joint1), ...] If users want to use interlocking joints, please overwrite this method.

joint_max_angles

joint_min_angles

larm

lleg

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

parent

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rarm

rleg

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

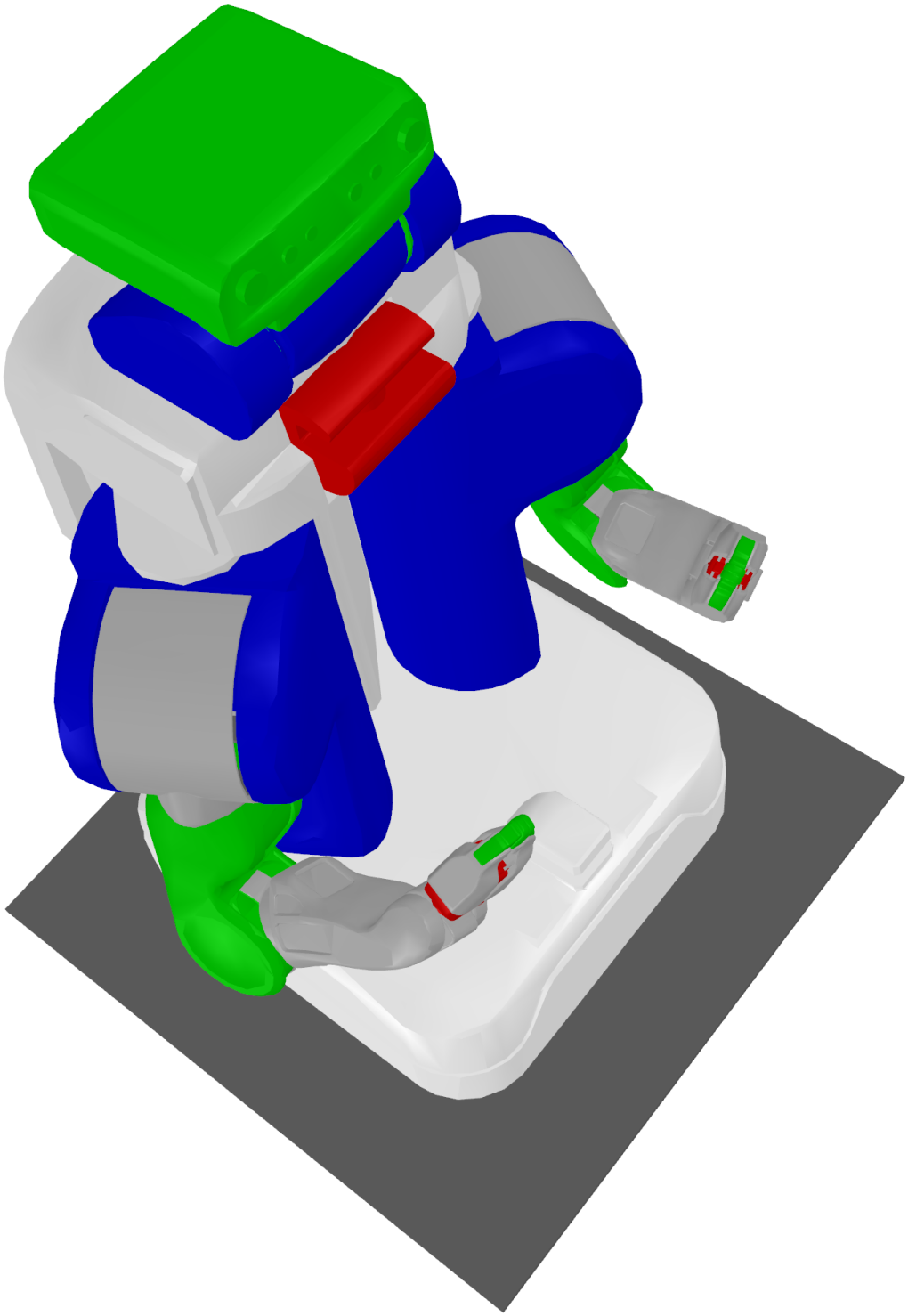
Return type
numpy.ndarray

z_axis

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray



3.3 Functions

3.3.1 Utilities functions

skrobot.coordinates.math._wrap_axis	Convert axis to float vector.
skrobot.coordinates.math._check_valid_rotation	Checks that the given rotation matrix is valid.
skrobot.coordinates.math._check_valid_translation	Checks that the translation vector is valid.
skrobot.coordinates.math.triple_product	Returns Triple Product
skrobot.coordinates.math.inverse_rodrigues	Inverse Rodrigues formula Convert Rotation-Matirx to Axis-Angle.
skrobot.coordinates.math.rotation_angle	Inverse Rodrigues formula Convert Rotation-Matirx to Axis-Angle.
skrobot.coordinates.math.make_matrix	Wrapper of numpy array.
skrobot.coordinates.math.random_rotation	Generates a random 3x3 rotation matrix.
skrobot.coordinates.math.random_translation	Generates a random translation vector.
skrobot.coordinates.math.midpoint	Return midpoint
skrobot.coordinates.math.midrot	Returns mid (or p) rotation matrix of given two matrix r1 and r2.
skrobot.coordinates.math.transform	Return transform m v
skrobot.coordinates.math.rotation_matrix	Return the rotation matrix.
skrobot.coordinates.math.rotate_vector	Rotate vector.
skrobot.coordinates.math.rotate_matrix	
skrobot.coordinates.math.rpy_matrix	Return rotation matrix from yaw-pitch-roll
skrobot.coordinates.math.rpy_angle	Decomposing a rotation matrix to yaw-pitch-roll.
skrobot.coordinates.math.normalize_vector	Return normalized vector
skrobot.coordinates.math.matrix_log	Returns matrix log of given rotation matrix, it returns [-pi, pi]
skrobot.coordinates.math.matrix_exponent	Returns exponent of given omega.
skrobot.coordinates.math.outer_product_matrix	Returns outer product matrix of given v.
skrobot.coordinates.math.rotation_matrix_from_rpy	Returns Rotation matrix from yaw-pitch-roll angles.
skrobot.coordinates.math.rotation_matrix_from_axis	Return rotation matrix orienting first_axis
skrobot.coordinates.math.rodrigues	Rodrigues formula.
skrobot.coordinates.math.rotation_angle	Inverse Rodrigues formula Convert Rotation-Matirx to Axis-Angle.
skrobot.coordinates.math.rotation_distance	Return the distance of rotation matrixes.
skrobot.coordinates.math.axis_angle_from_matrix	Converts the rotation of a matrix into axis-angle representation.
skrobot.coordinates.math.angle_between_vectors	Returns the smallest angle in radians between two vectors.

skrobot.coordinates.math._wrap_axis

skrobot.coordinates.math.**_wrap_axis**(*axis*)

Convert axis to float vector.

Parameters
axis (*list or numpy.ndarray or str or bool or None*) – rotation axis indicated by number or string.

Returns
axis – conveted axis

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import _wrap_axis
>>> _wrap_axis('x')
array([1, 0, 0])
>>> _wrap_axis('y')
array([0, 1, 0])
>>> _wrap_axis('z')
array([0, 0, 1])
>>> _wrap_axis('xy')
array([1, 1, 0])
>>> _wrap_axis([1, 1, 1])
array([1, 1, 1])
>>> _wrap_axis(True)
array([0, 0, 0])
>>> _wrap_axis(False)
array([1, 1, 1])
```

skrobot.coordinates.math._check_valid_rotation

skrobot.coordinates.math.**_check_valid_rotation**(*rotation*)

Checks that the given rotation matrix is valid.

skrobot.coordinates.math._check_valid_translation

skrobot.coordinates.math.**_check_valid_translation**(*translation*)

Checks that the translation vector is valid.

skrobot.coordinates.math.triple_product

skrobot.coordinates.math.**triple_product**(*a*, *b*, *c*)

Returns Triple Product

See https://en.wikipedia.org/wiki/Triple_product.

Geometrically, the scalar triple product

$a \cdot (b \times c)$

is the (signed) volume of the parallelepiped defined by the three vectors given.

Parameters

- **a** (*numpy.ndarray*) – vector a
- **b** (*numpy.ndarray*) – vector b
- **c** (*numpy.ndarray*) – vector c

Returns

triple product – calculated triple product

Return type

float

Examples

```
>>> from skrobot.math import triple_product
>>> triple_product([1, 1, 1], [1, 1, 1], [1, 1, 1])
0
>>> triple_product([1, 0, 0], [0, 1, 0], [0, 0, 1])
1
```

skrobot.coordinates.math.inverse_rodrigues

skrobot.coordinates.math.**inverse_rodrigues**(*mat*)

Inverse Rodrigues formula Convert Rotation-Matirx to Axis-Angle.

Return theta and axis. If given unit matrix, return None.

Parameters

mat (*numpy.ndarray*) – rotation matrix, shape (3, 3)

Returns

theta, axis – rotation angle in radian and rotation axis

Return type

tuple(float, numpy.ndarray)

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import rotation_angle
>>> rotation_angle(numpy.eye(3)) is None
True
>>> rotation_angle(numpy.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]]))
(1.5707963267948966, array([0., 1., 0.]))
```

skrobot.coordinates.math.rotation_angle

skrobot.coordinates.math.**rotation_angle**(*mat*)

Inverse Rodrigues formula Convert Rotation-Matirx to Axis-Angle.

Return theta and axis. If given unit matrix, return None.

Parameters
mat (*numpy.ndarray*) – rotation matrix, shape (3, 3)

Returns
theta, axis – rotation angle in radian and rotation axis

Return type
tuple(float, numpy.ndarray)

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import rotation_angle
>>> rotation_angle(numpy.eye(3)) is None
True
>>> rotation_angle(numpy.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]]))
(1.5707963267948966, array([0., 1., 0.]))
```

skrobot.coordinates.math.make_matrix

skrobot.coordinates.math.**make_matrix**(*r*, *c*)

Wrapper of numpy array.

Parameters

- **r** (*int*) – row of matrix
- **c** (*int*) – column of matrix

Returns
np.zeros((r, c), 'f') – matrix

Return type
numpy.ndarray

skrobot.coordinates.math.random_rotation

skrobot.coordinates.math.**random_rotation**()

Generates a random 3x3 rotation matrix.

Returns
rot – randomly generated 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import random_rotation
>>> random_rotation()
array([[-0.00933428, -0.90465681, -0.42603865],
       [-0.50305571, -0.36396787,  0.78387648],
       [-0.86420358,  0.2216381 , -0.4516954 ]])
>>> random_rotation()
array([[-0.6549113 ,  0.09499001, -0.749712  ],
       [-0.47962794, -0.81889635,  0.31522342],
       [-0.58399334,  0.5660262 ,  0.58186434]])
```

skrobot.coordinates.math.random_translation

skrobot.coordinates.math.**random_translation**()

Generates a random translation vector.

Returns
translation – A 3-entry random translation vector.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import random_translation
>>> random_translation()
array([0.03299473, 0.81481471, 0.57782565])
>>> random_translation()
array([0.10835455, 0.46549158, 0.73277675])
```

skrobot.coordinates.math.midpoint

skrobot.coordinates.math.**midpoint**(*p*, *a*, *b*)

Return midpoint

Parameters

- **p** (*float*) – ratio of a:b
- **a** (*numpy.ndarray*) – vector
- **b** (*numpy.ndarray*) – vector

Returns

midpoint – midpoint

Return type

numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import midpoint
>>> midpoint(0.5, np.ones(3), np.zeros(3))
>>> array([0.5, 0.5, 0.5])
```

skrobot.coordinates.math.midrot

skrobot.coordinates.math.**midrot**(*p*, *r1*, *r2*)

Returns mid (or p) rotation matrix of given two matrix r1 and r2.

Parameters

- **p** (*float*) – ratio of r1:r2
- **r1** (*numpy.ndarray*) – 3x3 rotation matrix
- **r2** (*numpy.ndarray*) – 3x3 rotation matrix

Returns

r – 3x3 rotation matrix

Return type

numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import midrot
>>> midrot(0.5,
        np.eye(3),
        np.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]]))
array([[ 0.70710678,  0.        ,  0.70710678],
       [ 0.        ,  1.        ,  0.        ],
       [-0.70710678,  0.        ,  0.70710678]])
```

(continues on next page)

(continued from previous page)

```
>>> from skrobot.coordinates.math import rpy_angle
>>> np.rad2deg(rpy_angle(midrot(0.5,
                np.eye(3),
                np.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]])))[0])
array([ 0., 45.,  0.])
```

skrobot.coordinates.math.transform

skrobot.coordinates.math.**transform**(*m*, *v*)

Return transform m v

Parameters

- **m** (*numpy.ndarray*) – 3 x 3 rotation matrix.
- **v** (*numpy.ndarray or list*) – input vector.

Returns

np.matmul(m, v) – transformed vector.

Return type

numpy.ndarray

skrobot.coordinates.math.rotation_matrix

skrobot.coordinates.math.**rotation_matrix**(*theta*, *axis*)

Return the rotation matrix.

Return the rotation matrix associated with counterclockwise rotation about the given axis by theta radians.

Parameters

- **theta** (*float*) – radian
- **axis** (*str or list or numpy.ndarray*) – rotation axis such that ‘x’, ‘y’, ‘z’ [0, 0, 1], [0, 1, 0], [1, 0, 0]

Returns

rot – rotation matrix about the given axis by theta radians.

Return type

numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import rotation_matrix
>>> rotation_matrix(np.pi / 2.0, [1, 0, 0])
array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  2.22044605e-16, -1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  2.22044605e-16]])
>>> rotation_matrix(np.pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

skrobot.coordinates.math.rotate_vector

skrobot.coordinates.math.**rotate_vector**(*vec*, *theta*, *axis*)

Rotate vector.

Rotate vec with respect to axis.

Parameters

- **vec** (*list or numpy.ndarray*) – target vector
- **theta** (*float*) – rotation angle
- **axis** (*list or numpy.ndarray or str*) – axis of rotation.

Returns

rotated_vec – rotated vector.

Return type

numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates.math import rotate_vector
>>> rotate_vector([1, 0, 0], pi / 6.0, [1, 0, 0])
array([1., 0., 0.])
>>> rotate_vector([1, 0, 0], pi / 6.0, [0, 1, 0])
array([ 0.8660254,  0.       , -0.5      ])
>>> rotate_vector([1, 0, 0], pi / 6.0, [0, 0, 1])
array([0.8660254, 0.5      , 0.       ])
```

skrobot.coordinates.math.rotate_matrix

skrobot.coordinates.math.**rotate_matrix**(*matrix*, *theta*, *axis*, *world=None*)

skrobot.coordinates.math.rpy_matrix

skrobot.coordinates.math.**rpy_matrix**(*az*, *ay*, *ax*)

Return rotation matrix from yaw-pitch-roll

This function creates a new rotation matrix which has been rotated ax radian around x-axis in WORLD, ay radian around y-axis in WORLD, and az radian around z axis in WORLD, in this order. These angles can be extracted by the rpy function.

Parameters

- **az** (*float*) – rotated around z-axis(yaw) in radian.
- **ay** (*float*) – rotated around y-axis(pitch) in radian.
- **ax** (*float*) – rotated around x-axis(roll) in radian.

Returns

r – rotation matrix

Return type

numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import rpy_matrix
>>> yaw = np.pi / 2.0
>>> pitch = np.pi / 3.0
>>> roll = np.pi / 6.0
>>> rpy_matrix(yaw, pitch, roll)
array([[ 1.11022302e-16, -8.66025404e-01,  5.00000000e-01],
       [ 5.00000000e-01,  4.33012702e-01,  7.50000000e-01],
       [-8.66025404e-01,  2.50000000e-01,  4.33012702e-01]])
```

skrobot.coordinates.math.rpy_angle

skrobot.coordinates.math.**rpy_angle**(*matrix*)

Decomposing a rotation matrix to yaw-pitch-roll.

Parameters
matrix (*list or numpy.ndarray*) – 3x3 rotation matrix

Returns
rpy – pair of rpy in yaw-pitch-roll order.

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import rpy_matrix
>>> from skrobot.coordinates.math import rpy_angle
>>> yaw = np.pi / 2.0
>>> pitch = np.pi / 3.0
>>> roll = np.pi / 6.0
>>> rot = rpy_matrix(yaw, pitch, roll)
>>> rpy_angle(rot)
(array([1.57079633, 1.04719755, 0.52359878]),
 array([ 4.71238898,  2.0943951 , -2.61799388]))
```

skrobot.coordinates.math.normalize_vector

skrobot.coordinates.math.**normalize_vector**(*v*, *ord=2*)

Return normalized vector

Parameters

- **v** (*list or numpy.ndarray*) – vector
- **ord** (*int (optional)*) – ord of np.linalg.norm

Returns
v – normalized vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import normalize_vector
>>> normalize_vector([1, 1, 1])
array([0.57735027, 0.57735027, 0.57735027])
>>> normalize_vector([0, 0, 0])
array([0., 0., 0.])
```

skrobot.coordinates.math.matrix_log

skrobot.coordinates.math.**matrix_log**(*m*)

Returns matrix log of given rotation matrix, it returns [-pi, pi]

Parameters
m (*list or numpy.ndarray*) – 3x3 rotation matrix

Returns
matrixlog – vector of shape (3,)

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import matrix_log
>>> matrix_log(np.eye(3))
array([0., 0., 0.])
```

skrobot.coordinates.math.matrix_exponent

skrobot.coordinates.math.**matrix_exponent**(*omega*, *p=1.0*)

Returns exponent of given omega.

This function is similar to cv2.Rodrigues. Convert rvec (which is log quaternion) to rotation matrix.

Parameters
omega (*list or numpy.ndarray*) – vector of shape (3,)

Returns
rot – exponential matrix of given omega

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import matrix_exponent
>>> matrix_exponent([1, 1, 1])
array([[ 0.22629564, -0.18300792,  0.95671228],
       [ 0.95671228,  0.22629564, -0.18300792],
       [-0.18300792,  0.95671228,  0.22629564]])
>>> matrix_exponent([1, 0, 0])
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.54030231, -0.84147098],
       [ 0.        ,  0.84147098,  0.54030231]])
```

skrobot.coordinates.math.outer_product_matrix

skrobot.coordinates.math.**outer_product_matrix**(*v*)

Returns outer product matrix of given v.

Returns following outer product matrix.

$$\begin{pmatrix} 0 & -v_2 & v_1 \\ v_2 & 0 & -v_0 \\ -v_1 & v_0 & 0 \end{pmatrix}$$

Parameters
v (*numpy.ndarray or list*) – [x, y, z]

Returns
matrix – 3x3 rotation matrix.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import outer_product_matrix
>>> outer_product_matrix([1, 2, 3])
array([[ 0, -3,  2],
       [ 3,  0, -1],
       [-2,  1,  0]])
```

skrobot.coordinates.math.rotation_matrix_from_rpy

skrobot.coordinates.math.**rotation_matrix_from_rpy**(*rpy*)

Returns Rotation matrix from yaw-pitch-roll angles.

Parameters
rpy (*numpy.ndarray or list*) – Vector of yaw-pitch-roll angles in radian.

Returns
rot – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import rotation_matrix_from_rpy
>>> rotation_matrix_from_rpy([0, np.pi / 3, 0])
array([[ 0.5      ,  0.       ,  0.8660254],
       [ 0.       ,  1.       ,  0.       ],
       [-0.8660254,  0.       ,  0.5      ]])
```

skrobot.coordinates.math.rotation_matrix_from_axis

skrobot.coordinates.math.**rotation_matrix_from_axis**(*first_axis=(1, 0, 0)*, *second_axis=(0, 1, 0)*, *axes='xy'*)

Return rotation matrix orienting first_axis

Parameters

- **first_axis** (*list or tuple or numpy.ndarray*) – direction of first axis
- **second_axis** (*list or tuple or numpy.ndarray*) – direction of second axis. This input axis is normalized using Gram-Schmidt.
- **axes** (*str*) – valid inputs are 'xy', 'yx', 'xz', 'zx', 'yz', 'zy'. first index indicates first_axis's axis.

Returns

rotation_matrix – Rotation matrix

Return type

numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import rotation_matrix_from_axis
>>> rotation_matrix_from_axis((1, 0, 0), (0, 1, 0))
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> rotation_matrix_from_axis((1, 1, 1), (0, 1, 0))
array([[ 0.57735027, -0.40824829, -0.70710678],
       [ 0.57735027,  0.81649658,  0.        ],
       [ 0.57735027, -0.40824829,  0.70710678]])
```

skrobot.coordinates.math.rodrigues

skrobot.coordinates.math.**rodrigues**(*axis*, *theta=None*)

Rodrigues formula.

See: Rodrigues' rotation formula - Wikipedia.

See: Axis-angle representation - Wikipedia.

Parameters

- **axis** (*numpy.ndarray or list*) – [x, y, z] vector. You can give axis-angle representation to *axis* if *theta* is None.
- **theta** (*float or None (optional)*) – radian. If None is given, calculate theta from axis.

Returns
mat – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import rodrigues
>>> rodrigues([1, 0, 0], 0)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> rodrigues([1, 1, 1], numpy.pi)
array([[-0.33333333,  0.66666667,  0.66666667],
       [ 0.66666667, -0.33333333,  0.66666667],
       [ 0.66666667,  0.66666667, -0.33333333]])
```

skrobot.coordinates.math.rotation_distance

skrobot.coordinates.math.**rotation_distance**(*mat1*, *mat2*)

Return the distance of rotation matrixes.

Parameters

- **mat1** (*list or numpy.ndarray*) –
- **mat2** (*list or numpy.ndarray*) – 3x3 matrix

Returns
diff_theta – distance of rotation matrixes in radian.

Return type
float

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import rotation_distance
>>> rotation_distance(numpy.eye(3), numpy.eye(3))
0.0
>>> rotation_distance(
        numpy.eye(3),
        numpy.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]]))
1.5707963267948966
```

skrobot.coordinates.math.axis_angle_from_matrix

skrobot.coordinates.math.**axis_angle_from_matrix**(*rotation*)

Converts the rotation of a matrix into axis-angle representation.

Parameters
rotation (*numpy.ndarray*) – 3x3 rotation matrix

Returns
axis_angle – axis-angle representation of vector

Return type
numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import axis_angle_from_matrix
>>> axis_angle_from_matrix(numpy.eye(3))
array([0, 0, 0])
>>> axis_angle_from_matrix(
    numpy.array([[0, 0, 1], [0, 1, 0], [-1, 0, 0]]))
array([0.        , 1.57079633, 0.        ])
```

skrobot.coordinates.math.angle_between_vectors

skrobot.coordinates.math.**angle_between_vectors**(*v1*, *v2*, *normalize=True*, *directed=True*)

Returns the smallest angle in radians between two vectors.

Parameters

- **v1** (*numpy.ndarray, list[float] or tuple(float)*) – input vector.
- **v2** (*numpy.ndarray, list[float] or tuple(float)*) – input vector.
- **normalize** (*bool*) – If normalize is True, normalize v1 and v2.
- **directed** (*bool*) – If directed is *False*, the input vectors are interpreted as undirected axes.

Returns
theta – smallest angle between v1 and v2.

Return type
float

3.3.2 Jacobian Functions

skrobot.coordinates.math.sr_inverse	Returns SR-inverse of given Jacobian.
skrobot.coordinates.math.sr_inverse_org	Return SR-inverse of given J
skrobot.coordinates.math.manipulability	Return manipulability of given matrix.

skrobot.coordinates.math.sr_inverse

skrobot.coordinates.math.**sr_inverse**(*J*, *k=1.0*, *weight_vector=None*)

Returns SR-inverse of given Jacobian.

Calculate Singularity-Robust Inverse See: *Inverse Kinematic Solutions With Singularity Robustness for Robot Manipulator Control*

Parameters

- **J** (*numpy.ndarray*) – jacobian
- **k** (*float*) – coefficients
- **weight_vector** (*None or numpy.ndarray*) – weight vector

Returns
ret – result of SR-inverse

Return type
numpy.ndarray

skrobot.coordinates.math.sr_inverse_org

skrobot.coordinates.math.**sr_inverse_org**(*J*, *k=1.0*)

Return SR-inverse of given J

Definition of SR-inverse is following.

$J^* = J^T(JJ^T + kI_m)^{-1}$

Parameters

- **J** (*numpy.ndarray*) – jacobian
- **k** (*float*) – coefficients

Returns
sr_inverse – calculated SR-inverse

Return type
numpy.ndarray

skrobot.coordinates.math.manipulability

skrobot.coordinates.math.**manipulability**(*J*)

Return manipulability of given matrix.

Definition of manipulability is following.

$w = \sqrt{\det J(\theta)J^T(\theta)}$

Parameters
J (*numpy.ndarray*) – jacobian

Returns
w – manipulability

Return type
float

3.3.3 Quaternion Functions

skrobot.coordinates.math.xyzw2wxyz	Convert quaternion [x, y, z, w] to [w, x, y, z] order.
skrobot.coordinates.math.wxyz2xyzw	Convert quaternion [w, x, y, z] to [x, y, z, w] order.
skrobot.coordinates.math.random_quaternion	Generate uniform random unit quaternion.
skrobot.coordinates.math.quaternion_multiply	Return multiplication of two quaternions.
skrobot.coordinates.math.quaternion_conjugate	Return conjugate of quaternion.
skrobot.coordinates.math.quaternion_inverse	Return inverse of quaternion.
skrobot.coordinates.math.quaternion_slerp	Return spherical linear interpolation between two quaternions.
skrobot.coordinates.math.quaternion_distance	Return the distance of quaternion.
skrobot.coordinates.math.quaternion_absolute_distance	Return the absolute distance of quaternion.
skrobot.coordinates.math.quaternion_norm	Return the norm of quaternion.
skrobot.coordinates.math.quaternion_normalize	Return the normalized quaternion.
skrobot.coordinates.math.matrix2quaternion	Returns quaternion of given rotation matrix.
skrobot.coordinates.math.quaternion2matrix	Returns matrix of given quaternion.
skrobot.coordinates.math.quaternion2rpy	Returns Roll-pitch-yaw angles of a given quaternion.
skrobot.coordinates.math.rpy2quaternion	Return Quaternion from yaw-pitch-roll angles.
skrobot.coordinates.math.rpy_from_quat	Returns Roll-pitch-yaw angles of a given quaternion.
skrobot.coordinates.math.quat_from_rotation_matrix	Returns quaternion of given rotation matrix.
skrobot.coordinates.math.quat_from_rpy	Return Quaternion from yaw-pitch-roll angles.
skrobot.coordinates.math.rotation_matrix_from_quat	Returns matrix of given quaternion.
skrobot.coordinates.math.quaternion_from_axis_angle	Return the quaternion from axis angle
skrobot.coordinates.math.axis_angle_from_quaternion	Converts a quaternion into the axis-angle representation.

skrobot.coordinates.math.xyzw2wxyz

skrobot.coordinates.math.**xyzw2wxyz**(*quat*)

Convert quaternion [x, y, z, w] to [w, x, y, z] order.

Parameters
quat (*list or numpy.ndarray*) – quaternion [x, y, z, w]

Returns
quaternion – quaternion [w, x, y, z]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import xyzw2wxyz
>>> xyzw2wxyz([1, 2, 3, 4])
array([4, 1, 2, 3])
```

skrobot.coordinates.math.wxyz2xyzw

skrobot.coordinates.math.**wxyz2xyzw**(*quat*)

Convert quaternion [w, x, y, z] to [x, y, z, w] order.

Parameters
quat (*list or numpy.ndarray*) – quaternion [w, x, y, z]

Returns
quaternion – quaternion [x, y, z, w]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import wxyz2xyzw
>>> wxyz2xyzw([1, 2, 3, 4])
array([2, 3, 4, 1])
```

skrobot.coordinates.math.random_quaternion

skrobot.coordinates.math.**random_quaternion**()

Generate uniform random unit quaternion.

Returns
quaternion – generated random unit quaternion [w, x, y, z]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import random_quaternion
>>> random_quaternion()
array([-0.02156994,  0.5404561 , -0.72781116, -0.42158374])
>>> random_quaternion()
array([-0.47302116,  0.020306  , -0.37539238,  0.79681818])
>>> from skrobot.coordinates.math import quaternion_norm
>>> q = random_quaternion()
>>> numpy.allclose(1.0, quaternion_norm(q))
True
>>> q.shape
(4,)
```

skrobot.coordinates.math.quaternion_multiply

skrobot.coordinates.math.**quaternion_multiply**(*quaternion1*, *quaternion0*)

Return multiplication of two quaternions.

Parameters

- **quaternion0** (*list or numpy.ndarray*) – [w, x, y, z]
- **quaternion1** (*list or numpy.ndarray*) – [w, x, y, z]

Returns

quaternion – [w, x, y, z]

Return type

numpy.ndarray

Examples

```
>>> q = quaternion_multiply([4, 1, -2, 3], [8, -5, 6, 7])
>>> numpy.allclose(q, [28, -44, -14, 48])
True
```

skrobot.coordinates.math.quaternion_conjugate

skrobot.coordinates.math.**quaternion_conjugate**(*quaternion*)

Return conjugate of quaternion.

Parameters

quaternion (*list or numpy.ndarray*) – quaternion [w, x, y, z]

Returns

conjugate of quaternion – [w, x, y, z]

Return type

numpy.ndarray

Examples

```
>>> q0 = random_quaternion()
>>> q1 = quaternion_conjugate(q0)
>>> np.allclose(quaternion_multiply(q0, q1), [1.0, 0, 0, 0])
True
```

skrobot.coordinates.math.quaternion_inverse

skrobot.coordinates.math.**quaternion_inverse**(*quaternion*)

Return inverse of quaternion.

Parameters
quaternion (*list or numpy.ndarray*) – [w, x, y, z]

Returns
inverse of quaternion – [w, x, y, z]

Return type
numpy.ndarray

Examples

```
>>> q0 = random_quaternion()
>>> q1 = quaternion_inverse(q0)
>>> np.allclose(quaternion_multiply(q0, q1), [1, 0, 0, 0])
True
```

skrobot.coordinates.math.quaternion_slerp

skrobot.coordinates.math.**quaternion_slerp**(*q0*, *q1*, *fraction*, *spin=0*, *shortestpath=True*)

Return spherical linear interpolation between two quaternions.

Parameters

- **q0** (*list or numpy.ndarray*) – start quaternion
- **q1** (*list or numpy.ndarray*) – end quaternion
- **fraction** (*float*) – ratio
- **spin** (*int*) – TODO
- **shortestpath** (*bool*) – TODO

Returns
quaternion – spherical linear interpolated quaternion

Return type
numpy.ndarray

Examples

```
>>> q0 = random_quaternion()
>>> q1 = random_quaternion()
>>> q = quaternion_slerp(q0, q1, 0.0)
>>> numpy.allclose(q, q0)
True
>>> q = quaternion_slerp(q0, q1, 1.0, 1)
>>> numpy.allclose(q, q1)
True
>>> q = quaternion_slerp(q0, q1, 0.5)
```

(continues on next page)

(continued from previous page)

```
>>> angle = math.acos(numpy.dot(q0, q))
>>> numpy.allclose(2.0, math.acos(numpy.dot(q0, q1)) / angle) or         numpy.
↪allclose(2.0, math.acos(-numpy.dot(q0, q1)) / angle)
True
```

skrobot.coordinates.math.quaternion_distance

skrobot.coordinates.math.**quaternion_distance**(*q1*, *q2*, *absolute=False*)

Return the distance of quaternion.

Parameters

- **q1** (*list or numpy.ndarray*) –
- **q2** (*list or numpy.ndarray*) – [w, x, y, z] order
- **absolute** (*bool*) – if True, return distance accounting for the sign ambiguity.

Returns

diff_theta – distance of q1 and q2 in radian.

Return type

float

Examples

```
>>> from skrobot.coordinates.math import quaternion_distance
>>> quaternion_distance([1, 0, 0, 0], [1, 0, 0, 0])
0.0
>>> quaternion_distance([1, 0, 0, 0], [0, 1, 0, 0])
3.141592653589793
>>> distance = quaternion_distance(
        [1, 0, 0, 0],
        [0.8660254, 0, 0.5, 0])
>>> np.rad2deg(distance)
60.00000021683236
```

skrobot.coordinates.math.quaternion_absolute_distance

skrobot.coordinates.math.**quaternion_absolute_distance**(*q1*, *q2*)

Return the absolute distance of quaternion.

Parameters

- **q1** (*list or numpy.ndarray*) –
- **q2** (*list or numpy.ndarray*) – [w, x, y, z] order

Returns

diff_theta – absolute distance of q1 and q2 in radian.

Return type

float

Examples

```
>>> from skrobot.coordinates.math import quaternion_absolute_distance
>>> quaternion_absolute_distance([1, 0, 0, 0], [1, 0, 0, 0])
0.0
>>> quaternion_absolute_distance(
        [1, 0, 0, 0],
        [0, 0.7071067811865476, 0, 0.7071067811865476])
3.141592653589793
>>> quaternion_absolute_distance(
        [-1, 0, 0, 0],
        [0, 0.7071067811865476, 0, 0.7071067811865476])
```

skrobot.coordinates.math.quaternion_norm

skrobot.coordinates.math.**quaternion_norm**(*q*)

Return the norm of quaternion.

Parameters
q (*list or numpy.ndarray*) – [w, x, y, z] order

Returns
norm_q – quaternion norm of q

Return type
float

Examples

```
>>> from skrobot.coordinates.math import quaternion_norm
>>> q = [1, 1, 1, 1]
>>> quaternion_norm(q)
2.0
>>> q = [0, 0.7071067811865476, 0, 0.7071067811865476]
>>> quaternion_norm(q)
1.0
```

skrobot.coordinates.math.quaternion_normalize

skrobot.coordinates.math.**quaternion_normalize**(*q*)

Return the normalized quaternion.

Parameters
q (*list or numpy.ndarray*) – [w, x, y, z] order

Returns
normalized_q – normalized quaternion

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import quaternion_normalize
>>> from skrobot.coordinates.math import quaternion_norm
>>> q = quaternion_normalize([1, 1, 1, 1])
>>> quaternion_norm(q)
1.0
```

skrobot.coordinates.math.matrix2quaternion

skrobot.coordinates.math.**matrix2quaternion**(*m*)

Returns quaternion of given rotation matrix.

Parameters
m (*list or numpy.ndarray*) – 3x3 rotation matrix

Returns
quaternion – quaternion [w, x, y, z] order

Return type
numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import matrix2quaternion
>>> matrix2quaternion(np.eye(3))
array([1., 0., 0., 0.])
```

skrobot.coordinates.math.quaternion2matrix

skrobot.coordinates.math.**quaternion2matrix**(*q*, *normalize=False*)

Returns matrix of given quaternion.

Parameters

- **quaternion** (*list or numpy.ndarray*) – quaternion [w, x, y, z] order
- **normalize** (*bool*) – if normalize is True, input quaternion is normalized.

Returns
rot – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import quaternion2matrix
>>> quaternion2matrix([1, 0, 0, 0])
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

skrobot.coordinates.math.quaternion2rpy

skrobot.coordinates.math.**quaternion2rpy**(*q*)

Returns Roll-pitch-yaw angles of a given quaternion.

Parameters
q (*numpy.ndarray or list*) – Quaternion in [w x y z] format.

Returns
rpy – Array of yaw-pitch-roll angles, in radian.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import quaternion2rpy
>>> quaternion2rpy([1, 0, 0, 0])
(array([ 0., -0.,  0.]), array([3.14159265, 3.14159265, 3.14159265]))
>>> quaternion2rpy([0, 1, 0, 0])
(array([ 0.        , -0.        ,  3.14159265]),
 array([3.14159265, 3.14159265, 0.        ]))
```

skrobot.coordinates.math.rpy2quaternion

skrobot.coordinates.math.**rpy2quaternion**(*rpy*)

Return Quaternion from yaw-pitch-roll angles.

Parameters
rpy (*numpy.ndarray or list*) – Vector of yaw-pitch-roll angles in radian.

Returns
quat – Quaternion in [w x y z] format.

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import rpy2quaternion
>>> rpy2quaternion([0, 0, 0])
array([1., 0., 0., 0.])
>>> yaw = np.pi / 3.0
>>> rpy2quaternion([yaw, 0, 0])
array([0.8660254, 0.       , 0.       , 0.5      ])
>>> rpy2quaternion([np.pi * 2 - yaw, 0, 0])
array([-0.8660254, -0.       ,  0.       ,  0.5      ])
```

skrobot.coordinates.math.rpy_from_quat

skrobot.coordinates.math.**rpy_from_quat**(*q*)

Returns Roll-pitch-yaw angles of a given quaternion.

Parameters
q (*numpy.ndarray or list*) – Quaternion in [w x y z] format.

Returns
rpy – Array of yaw-pitch-roll angles, in radian.

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import quaternion2rpy
>>> quaternion2rpy([1, 0, 0, 0])
(array([ 0., -0.,  0.]), array([3.14159265, 3.14159265, 3.14159265]))
>>> quaternion2rpy([0, 1, 0, 0])
(array([ 0.        , -0.        ,  3.14159265]),
 array([3.14159265, 3.14159265, 0.        ]))
```

skrobot.coordinates.math.quat_from_rotation_matrix

skrobot.coordinates.math.**quat_from_rotation_matrix**(*m*)

Returns quaternion of given rotation matrix.

Parameters
m (*list or numpy.ndarray*) – 3x3 rotation matrix

Returns
quaternion – quaternion [w, x, y, z] order

Return type
numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import matrix2quaternion
>>> matrix2quaternion(np.eye(3))
array([1., 0., 0., 0.])
```

skrobot.coordinates.math.quat_from_rpy

skrobot.coordinates.math.**quat_from_rpy**(*rpy*)

Return Quaternion from yaw-pitch-roll angles.

Parameters
rpy (*numpy.ndarray or list*) – Vector of yaw-pitch-roll angles in radian.

Returns
quat – Quaternion in [w x y z] format.

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates.math import rpy2quaternion
>>> rpy2quaternion([0, 0, 0])
array([1., 0., 0., 0.])
>>> yaw = np.pi / 3.0
>>> rpy2quaternion([yaw, 0, 0])
array([0.8660254, 0.       , 0.       , 0.5      ])
>>> rpy2quaternion([np.pi * 2 - yaw, 0, 0])
array([-0.8660254, -0.       ,  0.       ,  0.5      ])
```

skrobot.coordinates.math.rotation_matrix_from_quat

skrobot.coordinates.math.**rotation_matrix_from_quat**(*q*, *normalize=False*)

Returns matrix of given quaternion.

Parameters

- **quaternion** (*list or numpy.ndarray*) – quaternion [w, x, y, z] order
- **normalize** (*bool*) – if normalize is True, input quaternion is normalized.

Returns
rot – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import quaternion2matrix
>>> quaternion2matrix([1, 0, 0, 0])
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

skrobot.coordinates.math.quaternion_from_axis_angle

skrobot.coordinates.math.**quaternion_from_axis_angle**(*theta*, *axis*)

Return the quaternion from axis angle

This function returns quaternion associated with counterclockwise rotation about the given axis by theta radians.

Parameters

- **theta** (*float*) – radian
- **axis** (*list or numpy.ndarray*) – length is 3. Automatically normalize in this function

Returns

quaternion – [w, x, y, z] order

Return type

numpy.ndarray

Examples

```
>>> import numpy
>>> from skrobot.coordinates.math import quaternion_from_axis_angle
>>> quaternion_from_axis_angle(0.1, [1, 0, 0])
array([0.99875026, 0.04997917, 0.        , 0.        ])
>>> quaternion_from_axis_angle(numpy.pi, [1, 0, 0])
array([6.123234e-17, 1.000000e+00, 0.000000e+00, 0.000000e+00])
>>> quaternion_from_axis_angle(0, [1, 0, 0])
array([1., 0., 0., 0.])
>>> quaternion_from_axis_angle(numpy.pi, [1, 0, 1])
array([6.12323400e-17, 7.07106781e-01, 0.00000000e+00, 7.07106781e-01])
```

skrobot.coordinates.math.axis_angle_from_quaternion

skrobot.coordinates.math.**axis_angle_from_quaternion**(*quat*)

Converts a quaternion into the axis-angle representation.

Parameters

quat (*numpy.ndarray*) – quaternion [w, x, y, z]

Returns

axis_angle – axis-angle representation of vector

Return type

numpy.ndarray

Examples

```
>>> from skrobot.coordinates.math import axis_angle_from_quaternion
>>> axis_angle_from_quaternion([1, 0, 0, 0])
array([0, 0, 0])
>>> axis_angle_from_quaternion([0, 7.07106781e-01, 0, 7.07106781e-01])
array([2.22144147, 0.        , 2.22144147])
```

3.3.4 Geometry functions

skrobot.coordinates.geo.rotate_points	Rotate given points based on a starting and ending vector.

skrobot.coordinates.geo.rotate_points

skrobot.coordinates.geo.**rotate_points**(*points*, *a*, *b*)

Rotate given points based on a starting and ending vector.

Axis vector k is calculated from the any two nonzero vectors a and b. Rotated points are calculated from following Rodrigues rotation formula.

$$`P_{rot} = P\cos\theta + (k \times P)\sin\theta + k(k \cdot P)(1 - \cos\theta)`$$

Parameters

- **points** (*numpy.ndarray*) – Input points. The shape should be (3,) or (N, 3).
- **a** (*numpy.ndarray*) – nonzero vector.
- **b** (*numpy.ndarray*) – nonzero vector.

Returns

points_rot – rotated points.

Return type

numpy.ndarray

3.4 Interfaces

3.4.1 Pybullet Interface

You can use a Pybullet interface using skrobot.

skrobot.interfaces._pybullet.PybulletRobotInterface	Pybullet Interface Class

skrobot.interfaces._pybullet.PybulletRobotInterface

class skrobot.interfaces._pybullet.**PybulletRobotInterface**(*robot*, *urdf_path=None*, *use_fixed_base=False*, *connect=1*, **args*, ***kwargs*)

Pybullet Interface Class

Parameters

- **robot** (skrobot.model.RobotModel) – robot model
- **urdf_path** (*None or str*) – urdf path. If this value is *None*, get *urdf_path* from *robot.urdf_path*.
- **use_fixed_base** (*bool*) – If this value is *True*, robot in pybullet simulator will be fixed.
- **connect** (*int*) – pybullet's connection mode. If you have already connected to pybullet physics server, specify the server id. The default value is 1 (pybullet.GUI).

Examples

```
>>> from skrobot.models import PR2
>>> from skrobot.interfaces import PybulletRobotInterface
>>> robot_model = PR2()
>>> interface = PybulletRobotInterface(robot_model)
```

If you have already connected to pybullet physics server

```
>>> import pybullet
>>> client_id = pybullet.connect(pybullet.GUI)
>>> robot_model = PR2()
>>> interface = PybulletRobotInterface(robot_model, connect=client_id)
```

Methods

T()

Return 4x4 homogeneous transformation matrix.

Returns

matrix – homogeneous transformation matrix shape of (4, 4)

Return type

numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.T()
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
```

(continues on next page)

(continued from previous page)

```
       [0., 0., 0., 1.]])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.rotate(pi / 2.0, 'y')
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00,
         1.00000000e-01],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         2.00000000e-01],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16,
         3.00000000e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         1.00000000e+00]])
```

angle_vector(*angle_vector=None*, *realtime_simulation=None*)

Send a angle vector to pybullet's phsyics engine.

Parameters

- **angle_vector** (*None or numpy.ndarray*) – angle vector. If *None*, send self.robot.angle_vector()
- **realtime_simulation** (*None or bool*) – If this value is *True*, send angle_vector by pybullet.setJointMotorControl2.

Returns

angle_vector – return sent angle vector.

Return type

numpy.ndarray

static available()

Check Pybullet is available.

Returns

_available – If *False*, pybullet is not available.

Return type

bool

axis(*ax*)

changed()

Return False

This is used for CascadedCoords compatibility

Returns

False – always return False

Return type

bool

coords()

Return a deep copy of the Coordinates.

copy()

Return a deep copy of the Coordinates.

copy_coords()

Return a deep copy of the Coordinates.

copy_worldcoords()

Return a deep copy of the Coordinates.

difference_position(*coords*, *translation_axis=True*)

Return differences in positoin of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **translation_axis** (*str or bool or None (optional)*) – we can take 'x', 'y', 'z', 'xy', 'yz', 'zx', 'xx', 'yy', 'zz', True or False(None).

Returns

dif_pos – difference position of self coordinates and coords considering translation_axis.

Return type

numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates import transform_coords
>>> from numpy import pi
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(
...          pi / 3.0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 2.0, 'y')
>>> c1.difference_position(c2)
array([ 0.2       , -0.42320508,  0.3330127 ])
>>> c1 = Coordinates().translate([0.1, 0.2, 0.3]).rotate(0, 'x')
>>> c2 = Coordinates().translate([0.3, -0.3, 0.1]).rotate(
...          pi / 3.0, 'x')
>>> c1.difference_position(c2)
array([ 0.2, -0.5, -0.2])
```

difference_rotation(*coords*, *rotation_axis=True*)

Return differences in rotation of given coords.

Parameters

- **coords** (skrobot.coordinates.Coordinates) – given coordinates
- **rotation_axis** (*str or bool or None (optional)*) – we can take 'x', 'y', 'z', 'xx', 'yy', 'zz', 'xm', 'ym', 'zm', 'xy', 'yx', 'yz', 'zy', 'zx', 'xz', True or False(None).

Returns

dif_rot – difference rotation of self coordinates and coords considering rotation_axis.

Return type

numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import Coordinates
>>> from skrobot.coordinates.math import rpy_matrix
>>> coord1 = Coordinates()
>>> coord2 = Coordinates(rot=rpy_matrix(pi / 2.0, pi / 3.0, pi / 5.0))
>>> coord1.difference_rotation(coord2)
array([-0.32855112,  1.17434985,  1.05738936])
>>> coord1.difference_rotation(coord2, rotation_axis=False)
array([0, 0, 0])
>>> coord1.difference_rotation(coord2, rotation_axis='x')
array([0.        , 1.36034952, 0.78539816])
>>> coord1.difference_rotation(coord2, rotation_axis='y')
array([0.35398131, 0.        , 0.97442695])
>>> coord1.difference_rotation(coord2, rotation_axis='z')
array([-0.88435715,  0.74192175,  0.        ])
```

Using mirror option [‘xm’, ‘ym’, ‘zm’], you can allow differences of mirror direction.

```
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-2.99951957e-32,  0.00000000e+00,  0.00000000e+00])
>>> coord1 = Coordinates()
>>> coord2 = Coordinates().rotate(pi / 2.0, 'x')
>>> coord1.difference_rotation(coord2, 'xm')
array([-1.57079633,  0.        ,  0.        ])
```

disable_hook()

get_transform()

Return Transform object

Returns
transform – correnponding Transform to this coordinates

Return type
skrobot.coordinates.base.Transform

inverse_rotate_vector(*v*)

inverse_transform_vector(*vec*)

Transform vector in world coordinates to local coordinates

Parameters
vec (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

inverse_transformation(*dest=None*)

Return a invese transformation of this coordinate system.

Create a new coordinate with inverse transformation of this coordinate system.

$$\begin{pmatrix} R^{-1} & -R^{-1}p \\ 0 & 1 \end{pmatrix}$$

Parameters
dest (*None or* skrobot.coordinates.Coordinates) – If dest is given, the result of transformation is in-placed to dest.

Returns
dest – result of inverse transformation.

Return type
skrobot.coordinates.Coordinates

load_bullet()

Load bullet configurations.

This function internally called.

move_coords(*target_coords*, *local_coords*)

Transform this coordinate so that local_coords to target_coords.

Parameters

- **target_coords** (skrobot.coordinates.Coordinates) – target coords.
- **local_coords** (skrobot.coordinates.Coordinates) – local coords to be aligned.

Returns
self.worldcoords() – world coordinates.

Return type
skrobot.coordinates.Coordinates

newcoords(*c*, *pos=None*)

Update of position and orientation.

orient_with_matrix(*rotation_matrix*, *wrt='world'*)

Force update this coordinate system's rotation.

Parameters

- **rotation_matrix** (*numpy.ndarray*) – 3x3 rotation matrix.
- **wrt** (*str or* skrobot.coordinates.Coordinates) – reference coordinates.

parent_orientation(*v*, *wrt*)

rotate(*theta*, *axis=None*, *wrt='local'*)

Rotate this robot by given theta and axis.

For more detail, please see docs of skrobot.coordinates.Coordinates.rotate. The difference between the rotate, this function internally call pybullet.resetBasePositionAndOrientation.

Parameters

- **theta** (*float*) – radian
- **wrt** (*string or* skrobot.coordinates.Coordinates) –

rotate_vector(*v*)

Rotate 3-dimensional vector using rotation of this coordinate

Parameters
v (*numpy.ndarray*) – vector shape of (3,)

Returns
np.matmul(self.rotation, v) – rotated vector

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> from numpy import pi
>>> c = Coordinates().rotate(pi, 'z')
>>> c.rotate_vector([1, 2, 3])
array([-1., -2.,  3.])
```

rotate_with_matrix(*mat*, *wrt='local'*)

Rotate this coordinate by given rotation matrix.

This is a subroutine of self.rotate function.

Parameters

- **mat** (*numpy.ndarray*) – rotation matrix shape of (3, 3)
- **wrt** (*str or skrobot.coordinates.Coordinates*) – with respect to.

Returns
self

Return type
skrobot.coordinates.Coordinates

rpy_angle()

Return a pair of rpy angles of this coordinates.

Returns
rpy_angle(self.rotation) – a pair of rpy angles. See also skrobot.coordinates.math.rpy_angle

Return type
tuple(numpy.ndarray, numpy.ndarray)

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates().rotate(np.pi / 2.0, 'x').rotate(np.pi / 3.0, 'z')
>>> r.rpy_angle()
(array([ 3.84592537e-16, -1.04719755e+00,  1.57079633e+00]),
array([ 3.14159265, -2.0943951 , -1.57079633]))
```

sync()

Synchronize pybullet pose to robot_model.

transform(*c*, *wrt='local'*)

Transform this coordinate by coords based on wrt

For more detail, please see docs of skrobot.coordinates.Coordinates.transform. The difference between the transform, this function internally call pybullet.resetBasePositionAndOrientation.

Parameters

- **c** (skrobot.coordinates.Coordinates) – coordinate
- **wrt** (*string or* skrobot.coordinates.Coordinates) – If wrt is 'local' or self, multiply c from the right. If wrt is 'world' or 'parent' or self.parent, transform c with respect to worldcoord. If wrt is Coordinates, transform c with respect to c.

transform_vector(*v*)

"Return vector represented at world frame.

Vector v given in the local coords is converted to world representation.

Parameters
v (*numpy.ndarray*) – 3d vector. We can take batch of vector like (batch_size, 3)

Returns
transformed_point – transformed point

Return type
numpy.ndarray

transformation(*c2*, *wrt='local'*)

translate(*vec*, *wrt='local'*)

Translate robot in simulator.

For more detail, please see docs of skrobot.coordinates.Coordinates.translate. The difference between the translate, this function internally call pybullet.resetBasePositionAndOrientation.

Parameters

- **vec** (*list or np.ndarray*) – shape of (3,) translation vector. unit is [m] order.
- **wrt** (*string or* Coordinates *(optional)*) – translate with respect to wrt.

wait_interpolation(*thresh=0.05*, *timeout=60.0*)

Wait robot movement.

This function usually called after self.angle_vector. Wait while the robot joints are moving or until time of timeout. This function called internally pybullet.stepSimulation().

Parameters

- **thresh** (*float*) – velocity threshold for detecting movement stop.
- **timeout** (*float*) – maximum time of timeout.

worldcoords()

Return thisself

worldpos()

Return translation of this coordinate

See also skrobot.coordinates.Coordinates.translation

Returns
self.translation – translation of this coordinate

Return type
numpy.ndarray

worldrot()

Return rotation of this coordinate

See also skrobot.coordinates.Coordinates.rotation

Returns
self.rotation – rotation matrix of this coordinate

Return type
numpy.ndarray

__eq__(*value*, /)

Return self==value.

__ne__(*value*, /)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, /)

Return self>=value.

__mul__(*other_c*)

Return Transformed Coordinates.

Note that this function creates new Coordinates and does not change translation and rotation, unlike transform function.

Parameters
other_c (skrobot.coordinates.Coordinates) – input coordinates.

Returns
out – transformed coordinates multiplied other_c from the right. T = T_{self} T_{other_c}.

Return type
skrobot.coordinates.Coordinates

__pow__(*exponent*)

Return exponential homogeneous matrix.

If exponent equals -1, return inverse transformation of this coords.

Parameters
exponent (*numbers.Number*) – exponent value. If exponent equals -1, return inverse transformation of this coords. In current, support only -1 case.

Returns
out – output.

Return type
skrobot.coordinates.Coordinates

Attributes

dimension

Return dimension of this coordinate

Returns
len(self.translation) – dimension of this coordinate

Return type
int

dual_quaternion

Property of DualQuaternion

Return DualQuaternion representation of this coordinate.

Returns
DualQuaternion – DualQuaternion representation of this coordinate

Return type
skrobot.coordinates.dual_quaternion.DualQuaternion

name

Return this coordinate's name

Returns
self._name – name of this coordinate

Return type
str

pose

Getter of Pose in pybullet phsyics simulator.

Wrapper of pybullet.getBasePositionAndOrientation.

Returns
pose – pose of this robot in the phsyics simulator.

Return type
skrobot.coordinates.Coordinates

quaternion

Property of quaternion

Returns
q – [w, x, y, z] quaternion

Return type
numpy.ndarray

Examples

```
>>> from numpy import pi
>>> from skrobot.coordinates import make_coords
>>> c = make_coords()
>>> c.quaternion
array([1., 0., 0., 0.])
>>> c.rotate(pi / 3, 'y').rotate(pi / 5, 'z')
>>> c.quaternion
array([0.8236391 , 0.1545085 , 0.47552826, 0.26761657])
```

rotation

Return rotation matrix of this coordinates.

Returns
self._rotation – 3x3 rotation matrix

Return type
numpy.ndarray

Examples

```
>>> import numpy as np
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.rotation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> c.rotate(np.pi / 2.0, 'y')
>>> c.rotation
array([[ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  0.00000000e+00,  2.22044605e-16]])
```

translation

Return translation of this coordinates.

Returns
self._translation – vector shape of (3,). unit is [m]

Return type
numpy.ndarray

Examples

```
>>> from skrobot.coordinates import Coordinates
>>> c = Coordinates()
>>> c.translation
array([0., 0., 0.])
>>> c.translate([0.1, 0.2, 0.3])
>>> c.translation
array([0.1, 0.2, 0.3])
```

x_axis

Return x axis vector of this coordinates.

Returns
axis – x axis.

Return type
numpy.ndarray

y_axis

Return y axis vector of this coordinates.

Returns
axis – y axis.

Return type
numpy.ndarray

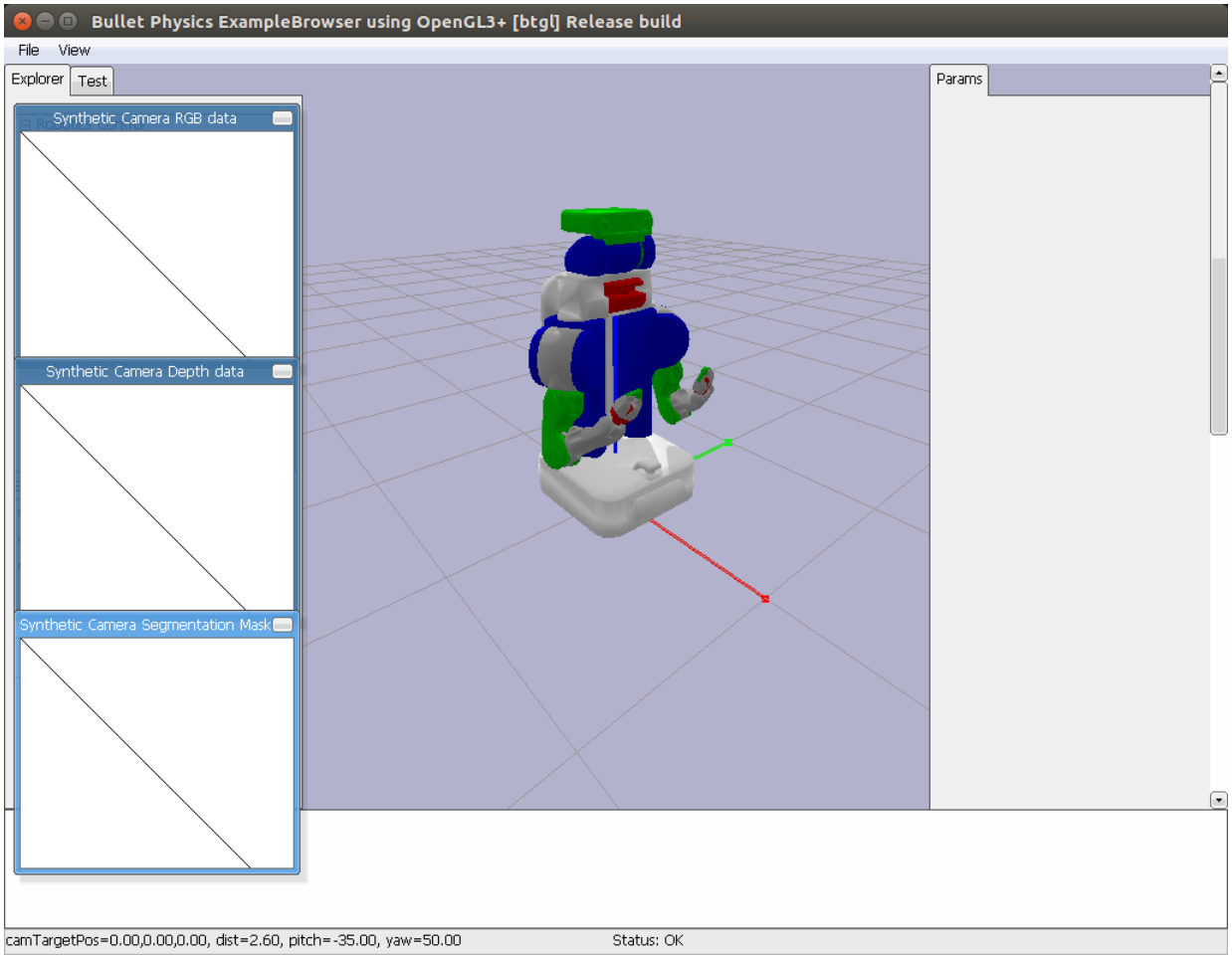
z_axis

Return z axis vector of this coordinates.

Returns
axis – z axis.

Return type
numpy.ndarray

```
>>> from skrobot.models import PR2
>>> from skrobot.interfaces import PybulletRobotInterface
>>> import pybullet
>>> client_id = pybullet.connect(pybullet.GUI)
>>> robot_model = PR2()
>>> interface = PybulletRobotInterface(robot_model, connect=client_id)
>>> interface.angle_vector(robot_model.reset_pose())
```


Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build
File
View
Explorer
Test
Synthetic Camera RGB data
Synthetic Camera Depth data
Synthetic Camera Segmentation Mask
Params
camTargetPos=0.00,0.00,0.00, dist=2.60, pitch=-35.00, yaw=50.00
Status: OK

3.5 Signed distance function (SDF)

3.5.1 SDF classes

skrobot.sdf.SignedDistanceFunction	A base class for signed distance functions (SDFs).
skrobot.sdf.UnionSDF	One can concat multiple SDFs *sdf_list* by using this class.
skrobot.sdf.BoxSDF	SDF for a box specified by *origin* and *width*.
skrobot.sdf.GridSDF	SDF using precopmuted signed distances for gridded points.
skrobot.sdf.CylinderSDF	SDF for a cylinder specified by *origin*,`radius` and *height*
skrobot.sdf.SphereSDF	SDF for a sphere specified by *origin* and *radius*.

skrobot.sdf.SignedDistanceFunction

class skrobot.sdf.**SignedDistanceFunction**(*origin*, *coords=None*, *use_abs=False*)

A base class for signed distance functions (SDFs).

Suffixes *_obj* and *_sdf* (e.g. in points_sdf) indicate that points are expressed in the sdf's object coordinate (*self.coords*) and sdf-specfic coordinate respectively. Each SDF performs the signed-distance computation in its own sdf-specific coordinate. For example, the origin of GridSDF's sdf-specific coordinate is the tip of the precomputed-gridded-box.

Usually, a child class implements the computation in the sdf-specific coordinates and SignedDistanceFunction wraps them so that the user can pass and get points and values expressed in an object coordinate. Thus, it is less likely that a user directly calls a method in a child class.

Methods

__call__(*points_obj*)

Compute signed distances of input points to the implicit surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. object coordinate.

Returns
signed_distances – 1 dim (n_point,) array of signed distance.

Return type
numpy.ndarray[float]

on_surface(*points_obj*)

Check if points are on the surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. an object coordinate.

Returns

- **logicals** (*numpy.ndarray[bool](n_point,)*) – boolean vector of the on-surface predicate for *points_obj*.

- **sd_vals** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to *logicals*.

surface_points(*n_sample=1000*)

Sample points from the implicit surface of the sdf.

Parameters
n_sample (*int*) – number of sample points.

Returns

- **points_obj** (*numpy.ndarray[float](n_point, 3)*) – sampled points w.r.t object coordinate.
- **dists** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to points_obj.

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)

Return self<=value.

__gt__(*value*, */*)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

skrobot.sdf.UnionSDF

class skrobot.sdf.**UnionSDF**(*sdf_list*, *coords=None*)

One can concat multiple SDFs *sdf_list* by using this class.

For consistency in the concatenation, it is required that the all SDFs to be concated are with *use_abs=False*.

Methods

__call__(*points_obj*)

Compute signed distances of input points to the implicit surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. object coordinate.

Returns
signed_distances – 1 dim (n_point,) array of signed distance.

Return type
numpy.ndarray[float]

classmethod from_robot_model(*robot_model*, *dim_grid=50*)

Create union sdf from a robot model

Parameters
robot_model (skrobot.model.RobotModel) – Using the links of the robot_model this creates the UnionSDF instance.

Returns
union_sdf – union sdf of robot_model

Return type
skrobot.sdf.UnionSDF

on_surface(*points_obj*)

Check if points are on the surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. an object coordinate.

Returns

- **logicals** (*numpy.ndarray[bool](n_point,)*) – boolean vector of the on-surface predicate for *points_obj*.
- **sd_vals** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to *logicals*.

surface_points(*n_sample=1000*)

Sample points from the implicit surface of the sdf.

Parameters
n_sample (*int*) – number of sample points.

Returns

- **points_obj** (*numpy.ndarray[float](n_point, 3)*) – sampled points w.r.t object coordinate.
- **dists** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to points_obj.

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)

Return self<=value.

__gt__(*value*, */*)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

skrobot.sdf.BoxSDF

class skrobot.sdf.**BoxSDF**(*origin*, *width*, *coords=None*, *use_abs=False*)

SDF for a box specified by *origin* and *width*.

Methods

__call__(*points_obj*)

Compute signed distances of input points to the implicit surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. object coordinate.

Returns
signed_distances – 1 dim (n_point,) array of signed distance.

Return type
numpy.ndarray[float]

on_surface(*points_obj*)

Check if points are on the surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. an object coordinate.

Returns

- **logicals** (*numpy.ndarray[bool](n_point,)*) – boolean vector of the on-surface predicate for *points_obj*.
- **sd_vals** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to *logicals*.

surface_points(*n_sample=1000*)

Sample points from the implicit surface of the sdf.

Parameters
n_sample (*int*) – number of sample points.

Returns

- **points_obj** (*numpy.ndarray[float](n_point, 3)*) – sampled points w.r.t object coordinate.
- **dists** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to points_obj.

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)

Return self<=value.

__gt__(*value*, */*)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

skrobot.sdf.GridSDF

class skrobot.sdf.**GridSDF**(*sdf_data*, *origin*, *resolution*, *fill_value=inf*, *coords=None*, *use_abs=False*)

SDF using precopmuted signed distances for gridded points.

Methods

__call__(*points_obj*)

Compute signed distances of input points to the implicit surface.

Parameters

points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. object coordinate.

Returns

signed_distances – 1 dim (n_point,) array of signed distance.

Return type

numpy.ndarray[float]

static **from_file**(*filepath*, ***kwargs*)

Return GridSDF instance from a .sdf file.

Parameters

filepath (*str or pathlib.Path*) – path of .sdf file

Returns

sdf_instance – instance of sdf

Return type

skrobot.esdf.GridSDF

static **from_objfile**(*obj_filepath*, *dim_grid=100*, *padding_grid=5*, ***kwargs*)

Return GridSDF instance from an .obj file.

In the initial call of this method for an .obj file, the pre-process takes some time to converting it to a .sdf file. However, because a cache of .sdf file is created in the initial call, there is no overhead from the next call for the same .obj file.

Parameters

- **obj_filepath** (*str or pathlib.Path*) – path of objfile
- **dim_grid** (*int*) – dim of sdf
- **padding_grid** (*int*) – number of padding

Returns

sdf_instance – instance of sdf

Return type

skrobot.sdf.GridSDF

is_out_of_bounds(*points_obj*)

check if the the input points is out of bounds

This method checks if the the input points is out of bounds of RegularGridInterpolator.

Parameters
points_obj (*numpy.ndarray[float](n_points, 3)*) – points w.r.t. object to be checked.

Returns
is_out_arr – If points is out of the interpolator's boundary, the correspoinding element of is_out_arr is True

Return type
numpy.ndarray[bool](n_points,)

on_surface(*points_obj*)

Check if points are on the surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. an object coordinate.

Returns

- **logicals** (*numpy.ndarray[bool](n_point,)*) – boolean vector of the on-surface predicate for *points_obj*.
- **sd_vals** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to *logicals*.

surface_points(*n_sample=1000*)

Sample points from the implicit surface of the sdf.

Parameters
n_sample (*int*) – number of sample points.

Returns

- **points_obj** (*numpy.ndarray[float](n_point, 3)*) – sampled points w.r.t object coordinate.
- **dists** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to points_obj.

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)

Return self<=value.

__gt__(*value*, */*)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

skrobot.sdf.CylinderSDF

class skrobot.sdf.**CylinderSDF**(*origin*, *height*, *radius*, *coords=None*, *use_abs=False*)

SDF for a cylinder specified by *origin*,`radius` and *height*

Methods

__call__(*points_obj*)

Compute signed distances of input points to the implicit surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. object coordinate.

Returns
signed_distances – 1 dim (n_point,) array of signed distance.

Return type
numpy.ndarray[float]

on_surface(*points_obj*)

Check if points are on the surface.

Parameters
points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. an object coordinate.

Returns

- **logicals** (*numpy.ndarray[bool](n_point,)*) – boolean vector of the on-surface predicate for *points_obj*.
- **sd_vals** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to *logicals*.

surface_points(*n_sample=1000*)

Sample points from the implicit surface of the sdf.

Parameters
n_sample (*int*) – number of sample points.

Returns

- **points_obj** (*numpy.ndarray[float](n_point, 3)*) – sampled points w.r.t object coordinate.
- **dists** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to points_obj.

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)

Return self<=value.

__gt__(*value*, */*)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

skrobot.sdf.SphereSDF

class skrobot.sdf.**SphereSDF**(*origin*, *radius*, *coords=None*, *use_abs=False*)

SDF for a sphere specified by *origin* and *radius*.

Methods

__call__(*points_obj*)

Compute signed distances of input points to the implicit surface.

Parameters

points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. object coordinate.

Returns

signed_distances – 1 dim (n_point,) array of signed distance.

Return type

numpy.ndarray[float]

on_surface(*points_obj*)

Check if points are on the surface.

Parameters

points_obj (*numpy.ndarray[float](n_point, 3)*) – 2 dim point array w.r.t. an object coordinate.

Returns

- **logicals** (*numpy.ndarray[bool](n_point,)*) – boolean vector of the on-surface predicate for *points_obj*.
- **sd_vals** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to *logicals*.

surface_points(*n_sample=1000*)

Sample points from the implicit surface of the sdf.

Parameters

n_sample (*int*) – number of sample points.

Returns

- **points_obj** (*numpy.ndarray[float](n_point, 3)*) – sampled points w.r.t object coordinate.
- **dists** (*numpy.ndarray[float](n_point,)*) – signed distances corresponding to points_obj.

__eq__(*value*, */*)

Return self==value.

__ne__(*value*, */*)

Return self!=value.

__lt__(*value*, */*)

Return self<value.

__le__(*value*, */*)
Return self<=value.

__gt__(*value*, */*)
Return self>value.

__ge__(*value*, */*)
Return self>=value.

3.5.2 SDF utilities

skrobot.sdf.signed_distance_function.trimesh2sdf	Convert trimesh to signed distance function.
skrobot.sdf.signed_distance_function.link2sdf	Convert Link to corresponding sdf

skrobot.sdf.signed_distance_function.trimesh2sdf

skrobot.sdf.signed_distance_function.**trimesh2sdf**(*mesh*, *dim_grid=100*, *padding_grid=5*)
Convert trimesh to signed distance function.

Parameters

- **mesh** (*trimesh.base.Trimesh*) – mesh object.
- **dim_grid** (*int*) – dimension of the GridSDF. This value is used for a not primitive mesh.
- **padding_grid** (*int*) – number of padding.

Returns
sdf – converted signed distance function.

Return type
skrobot.sdf.SignedDistanceFunction

skrobot.sdf.signed_distance_function.link2sdf

skrobot.sdf.signed_distance_function.**link2sdf**(*link*, *urdf_path*, *dim_grid=30*)
Convert Link to corresponding sdf

Parameters

- **link** (*skrobot.model.Link*) – link object
- **urdf_path** (*str*) – urdf path of the robot model that the link belongs to
- **dim_grid** (*int*) – dimension of the GridSDF

Returns
sdf – corresponding signed distance function to the link type. e.g. if Link has geometry of urdf.Box, then BoxSDF is created.

Return type
skrobot.sdf.SignedDistanceFunction

3.6 Planning

3.6.1 Sdf-swept-sphere-based collision checker

skrobot.planner.SweptSphereSdfCollisionChecker	Collision checker between swept spheres and sdf

skrobot.planner.SweptSphereSdfCollisionChecker

class skrobot.planner.**SweptSphereSdfCollisionChecker**(*sdf*, *robot_model*)

Collision checker between swept spheres and sdf

Methods

add_coll_spheres_to_viewer(*viewer*)

Add collision sheres to viewer

Parameters
viewer (*skrobot.viewers._trimesh.TrimeshSceneViewer*) – viewer

add_collision_link(*coll_link*)

Add link for which collision with sdf is checked

The given *coll_link* will be approximated by swept-spheres and these spheres will be added to collision sphere's list.

Parameters
coll_link (*skrobot.model.Link*) – link for which collision with sdf is checked

add_collision_links(*coll_links*)

Add links for which collisions with SDF is checked.

The given *coll_links* will be approximated by swept-spheres and these spheres will be added to collision sphere's list.

Parameters
coll_links (*list[skrobot.model.Link]*) – link list for which collisions with sdf is checked.

collision_check()

Check collision between links and collision spheres.

Returns
is_collision – *True* if a collision occurred between any pair of links and collision spheres and *False* otherwise.

Return type
bool

compute_batch_sd_vals(*joint_list*, *angle_vector_seq*, *with_base=False*, *with_jacobian=False*)

Compute sd signed distances of collision spheres

This method is the core of this class. We assume that this method is mainly called from a trajecotyr optimizer or path planner.

Let n_{wp} be the number of way-points of a trajectory. Let n_f be the number of collision feature points on the robot links.

Let $f_{i,j} : \mathbb{R}^{n_{dof}} \ni q \mapsto x \in \mathbb{R}^3$ be the forward kinematics of collision features point j at waypoint i where q is the angle vector and n_{dof} is the dimension of the angle vector.

Let $c : \mathbb{R}^3 \ni x \mapsto sd \in \mathbb{R}$ be the signd distance function.

Then, this fucntion is defined as $F : \mathbb{R}^{n_{wp} n_{dof}} \ni \xi \mapsto [[f_{1,1}(q_1), \ldots, f_{1,n_f}(q_1)]^T, \ldots, [f_{n_{wp},1}(q_{n_{wp}}), \ldots, f_{n_{wp},n_f}(q_{n_{wp}})]^T]^T \in \mathbb{R}^{n_{wp} n_f}$ where $\xi = [q_1^T, \ldots, q_{n_{wp}}^T]^T$ be the angle vector sequence of the trajectory. The corresponding jacobian is defined as $\frac{\partial F}{\partial \xi}$.

Parameters

- **joint_list** (*list[skrobot.model.Joint]*) – joint list to be set
- **angle_vector_seq** (*numpy.ndarray[float](n_wp, n_dof)*) – angle vector sequence.
- **with_base** (*bool*) – hoge
- **with_jacobian** (*bool*) – if *True*, jacobian is copmuted at the same time.

Returns

- **sd_vals** (*numpy.ndarray[float](n_wp * n_feature,)*) – signed distnaces for all feature points through the trajectory
- **sd_vals_jacobi** (*numpy.ndarray[float](n_wp * n_feature, n_wp * n_dof)*) – jacobain of sd_vals with respect to DOF (i.e. n_wp * n_dof) of the trajectory

delete_coll_spheres_from_viewer(*viewer*)

Delete collision sheres from viewer

Parameters
viewer (*skrobot.viewers._trimesh.TrimeshSceneViewer*) – viewer

update_color()

Update the color of links under collision

This method checks the collision between the collision spheres and registered sdf. If collision spheres are found to be under collision, the color of the spheres will be changed to *color_collision_sphere*.

Returns
dists – array of the signed distances for each sphere against sdf.

Return type
numpy.ndarray(n_sphere,)

__eq__(*value*, /)

Return self==value.

__ne__(*value*, /)

Return self!=value.

__lt__(*value*, /)

Return self<value.

__le__(*value*, /)

Return self<=value.

__gt__(*value*, /)

Return self>value.

__ge__(*value*, */*)

Return self>=value.

Attributes

n_feature

Return number of collision sphere.

Returns

n_feature – number of collision spheres.

Return type

int

3.6.2 SQP-based trajectory planner

skrobot.planner.sqp_plan_trajectory	Gradient based trajectory optimization using scipy's SLSQP.

skrobot.planner.sqp_plan_trajectory

skrobot.planner.**sqp_plan_trajectory**(*collision_checker*, *av_start*, *av_goal*, *joint_list*, *n_wp*, *safety_margin=0.01*, *with_base=False*, *weights=None*, *initial_trajectory=None*, *slsqp_option=None*)

Gradient based trajectory optimization using scipy's SLSQP.

Collision constraint is considered in an inequality constraits. Terminal constraint (start and end) is considered as an equality constraint.

Parameters

- **av_start** (*numpy.ndarray(n_dof,)*) – joint angle vector at start point
- **joint_list** (*list[skrobot.model.Link]*) – link list to be controlled (similar to inverse_kinematics function)
- **n_wp** (*int*) – number of waypoints
- **safety_margin** (*float*) – safety margin in collision checking
- **with_base** (*bool*) – If *with_base=False*, *n_dof* is the number of joints *n_joint*, but if *with_base=True*, *n_dof = len(joint_list) + 3*.
- **weights** (*numpy.ndarray(n_dof,) or None*) – cost to move of each joint. For example, if you set weights=numpy.ndarray([1.0, 0.1, 0.1]) for a 3 DOF manipulator, moving the first joint is with high cost compared to others. If set to *None* it's automatically determined.
- **initial_trajectory** (*numpy.ndarray(n_wp, n_dof) or None*) – initial solution in the trajectory optimization specifeid by a angle vector sequence. If None, initial trajectory is automatically generated.
- **slsqp_option** (*dict or None*) – option of slsqp. Please see *options* in https://docs.scipy.org/doc/scipy/reference/optimize.minimize-slsqp.html for the detail. If set to *None*, a default values is used.

Returns
planned_trajectory – planned trajectory.

Return type
numpy.ndarray(n_wp, n_dof)

3.6.3 Swept sphere generater

skrobot.planner.swept_sphere.compute_swept_sphere	Compute swept spheres approximating a mesh

skrobot.planner.swept_sphere.compute_swept_sphere

skrobot.planner.swept_sphere.**compute_swept_sphere**(*collision_mesh*, *n_sphere=None*, *tol=0.1*)

Compute swept spheres approximating a mesh

Parameters

- **collision_mesh** (*trimesh.Trimesh*) – mesh which swept spheres are computed for
- **n_sphere** (*int or None*) – number of sphere to approximate the mesh. If it's set to *None*, the number of sphere is automatically determined.
- **tol** (*float*) – tolerance determins how much mesh jutting-out from the swept-spheres are accepted. Let *max_jut* be the maximum jut-distance. Then the setting *tol* enforces *max_jut / radius* < *max_jut*. If some integer is set to *n_sphere*, *tol* does not affects the result.

Returns

- **centers_original_space** (*numpy.ndarray[float](n_sphere, 3)*) – center of the approximating spheres in the space where the mesh vertices are defined.
- **radius** (*float*) – radius of approximating sphers.

3.6.4 Planner utils

skrobot.planner.utils.scipinize	Scipinize a function returning both f and jac
skrobot.planner.utils.set_robot_config	A utility function for setting robot state
skrobot.planner.utils.get_robot_config	A utility function for getting robot state
skrobot.planner.utils.forward_kinematics_multi	Compute fk for multiple feature points

skrobot.planner.utils.scipinize

skrobot.planner.utils.**scipinize**(*fun*)

Scipinize a function returning both f and jac

For the detail this issue may help: https://github.com/scipy/scipy/issues/12692

Parameters
fun (*function*) – function maps numpy.ndarray(n_dim,) to tuple[numpy.ndarray(m_dim,), numpy.ndarray(m_dim, n_dim)], where the returned tuples is composed of function value(vector) and the corresponding jacobian.

Returns

- **fun_scipinized** (*function*) – function maps numpy.ndarray(n_dim,) to a value numpy.ndarray(m_dim,).
- **fun_scipinized_jac** (*function*) – function maps numpy.ndarray(n_dim,) to jacobian numpy.ndarray(m_dim, n_dim).

skrobot.planner.utils.set_robot_config

skrobot.planner.utils.**set_robot_config**(*robot_model*, *joint_list*, *av*, *with_base=False*)

A utility function for setting robot state

Parameters

- **robot_model** (*skrobot.model.CascadedLink*) – robot model
- **joint_list** (*list[skrobot.model.Joint]*) – joint list to be set
- **av** (*numpy.ndarray[float](n_dof,)*) – angle vector which has n_dof dims.
- **with_base** (*bool*) – If *with_base=False*, *n_dof* is the number of joints *n_joint*, but if *with_base=True*, *n_dof* = *n_joint* + *3*.

skrobot.planner.utils.get_robot_config

skrobot.planner.utils.**get_robot_config**(*robot_model*, *joint_list*, *with_base=False*)

A utility function for getting robot state

Parameters

- **robot_model** (*skrobot.model.CascadedLink*) – robot model
- **joint_list** (*list[Joint]*) – joint list of which you want to know the angles
- **with_base** (*bool*) – If set to *True*, base position is also computed.

Returns

av_whole (or av_joint) – angle vector. If *with_base=False*, *n_dof* is the number of joints *n_joint*, but if *with_base=True*, *n_dof* = *n_joint* + *3*.

Return type

numpy.ndarray(n_dof,)

skrobot.planner.utils.forward_kinematics_multi

skrobot.planner.utils.**forward_kinematics_multi**(*robot_model*, *joint_list*, *av*, *move_target_list*, *with_rot*, *with_base*, *with_jacobian*)

Compute fk for multiple feature points

Parameters

- **robot_model** (*skrobot.model.CascadedLink*) – robot model.
- **joint_list** (*list[skrobot.model.Joint]*) – joint to be controlled
- **av** (*numpy.ndarray(n_dof,)*) – angle vector.
- **move_target_list** (*list[skrobot.coordinates.CascadedCoords]*) – the list has *n_feature* elements. Each element is the coordinate of the features points.

- **with_rot** (*bool*) – If set to *True*, 7(3 + 4) dim pose-fk is also computed. Otherwise, 3 dim point-fk is computed.
- **with_base** (*bool*) – If *with_base=False*, *n_dof* is the number of joints *n_joint*, but if *with_base=True*, *n_dof = n_joint + 3*.

Returns

- **pose_arr** (*numpy.ndarray(n_feature, dim_pose)*) – array of pose of each feature points. *dim_pose=7' if `with_rot=True*. Otherwise, *dim_pose=3*.
- **jac_arr** (*numpy.ndarray(n_feature, dim_pose, n_dof)*) – array of jacobian of each feature points.

CHAPTER

FOUR

DEVELOPMENT GUIDE

Read this guide before doing development in `skrobot`.

4.1 Setting Up

To set up the tools you'll need for developing, you'll need to install `skrobot` in development mode. Start by installing the development dependencies:

```
git clone https://github.com/iory/scikit-robot.git
cd scikit-robot
pip install -e .
```

4.2 Running Code Style Checks

We follow PEP 8 and partially OpenStack Style Guidelines as basic style guidelines. Any contributions in terms of code are expected to follow these guidelines.

You can use the `autopep8`, `isort` and the `flake8` commands to check whether or not your code follows the guidelines. In order to avoid confusion from using different tool versions, we pin the versions of those tools. Install them with the following command (from within the top directory of the Chainer repository):

```
$ pip install hacking pytest autopep8 isort
```

And check your code with:

```
$ autopep8 path/to/your/code.py
$ flake8 path/to/your/code.py
```

`autopep8` can automatically correct Python code to conform to the PEP 8 style guide:

```
$ autopep8 --in-place path/to/your/code.py
```

`isort` can automatically correct `import order`:

```
$ cd scikit-robot && isort path/to/your/code.py
```

For more information, please see the flake8 documentation.

4.3 Running Tests

This project uses pytest, the standard Python testing framework. Their website has tons of useful details, but here are the basics.

To run the testing suite, simply navigate to the top-level folder in `scikit-robot` and run the following command:

```
pytest -v tests
```

You should see the testing suite run. There are a few useful command line options that are good to know:

- `-s` - Shows the output of `stdout`. By default, this output is masked.
- `--pdb` - Instead of crashing, opens a debugger at the first fault.
- `--lf` - Instead of running all tests, just run the ones that failed last.
- `--trace` - Open a debugger at the start of each test.

You can see all of the other command-line options here.

By default, `pytest` will look in the `tests` folder recursively. It will run any function that starts with `test_` in any file that starts with `test_`. You can run `pytest` on a directory or on a particular file by specifying the file path:

```
pytest -v tests/skrobot_tests/coordinates_tests/test_math.py
```

4.4 Building Documentation

To build `scikit-robot`'s documentation, go to the `docs` directory and run `make` with the appropriate target. For example,

```
cd docs/
make html
```

will generate HTML-based docs, which are probably the easiest to read. The resulting index page is at `docs/build/html/index.html`. If the docs get stale, just run `make clean` to remove all build files.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z