rigid-body-motion Documentation

Release 0.9.1

Peter Hausamann

Aug 06, 2023

GETTING STARTED

Python utilities for estimating and transforming rigid body motion.

Hosted on GitHub: https://github.com/phausamann/rigid-body-motion

CHAPTER

ONE

OVERVIEW

This package provides a high-level interface for transforming arrays describing motion of rigid bodies between different coordinate systems and reference frames. The core of the reference frame handling is a fast re-implementation of ROS's `tf2` library using `numpy` and `numpy-quaternion`. The package also provides first-class support for xarray data types.

1.1 Installation

1.1.1 Latest version

rigid-body-motion can be installed via `pip`:

```
$ pip install rigid-body-motion
```

or via `conda`:

```
$ conda install -c phausamann -c conda-forge rigid-body-motion
```

1.1.2 Optional dependencies

Optional dependencies can be installed via `pip` or `conda` (just replace `pip install` with `conda install`).

Transformations can be sped up significantly by installing the numba library:

```
$ pip install numba
```

rigid-body-motion supports xarray data types:

```
$ pip install xarray
```

Loading the example datasets requires the pooch and netCDF4 libraries:

```
$ pip install pooch netcdf4
```

Plotting functions require matplotlib:

```
$ pip install matplotlib
```

1.1.3 ROS integration

The package also integrates with ROS, which can be installed quite conveniently via `conda` thanks to the amazing work by the people from RoboStack. This even works on Windows and macOS! The necessary dependencies are:

```
$ conda install -c conda-forge -c robostack ros-noetic-rospy ros-noetic-tf ros-noetic-
↪tf2-ros ros-noetic-tf2-geometry-msgs ros-noetic-geometry-msgs ros-noetic-visualization-
↪msgs python-orocos-kdl boost=1.74
```

Note that these packages are only available for Python 3.6 and 3.8.

In order to use RViz for visualization you need to install that as well:

```
$ conda install -c conda-forge -c robostack ros-noetic-rviz
```

Since ROS can communicate across conda environments, we would however recommend installing RViz in a dedicated environment:

```
$ conda create -n ros -c conda-forge -c robostack ros-noetic-rviz
```

After installing, you can spin up a `roscore`:

```
$ conda activate ros
$ roscore
```

and, from a second terminal, launch RViz:

```
$ conda activate ros
$ rviz
```

Troubleshooting

In order to make sure all ROS dependencies have been set up correctly, run:

```
$ python -m rigid_body_motion.ros.check_install
```

If this is not successful, you need to fix some of your dependencies. The following outlines solutions for common issues:

1. Problem: boost version

   ```
   ImportError: libboost_thread.so.1.74.0: cannot open shared object file:
   No such file or directory
   ```

   Solution: install correct boost version

   ```
   $ conda install -c conda-forge boost=1.74
   ```

   Replace `1.74` with whatever version the traceback tells you is missing.

1.1.4 Example environments

We recommend using rigid_body_motion in a dedicated conda environment. For the examples in the user guide, we provide an `environment file` that you can download and set up with:

```
$ conda env create -f example-env.yml
```

There's also an example environment that includes the necessary ROS packages (`here`) that you can set up with:

```
$ conda env create -f example-env-ros.yml
```

If you download the examples as Jupyter notebooks, it is sufficient to have the Jupyter notebook server installed in your base environment alongside `nb_conda`:

```
$ conda install -n base nb_conda
```

Now you can open any of the example notebooks, go to *Kernel* > *Change kernel* and select *Python [conda env:rbm-examples]* (or *Python [conda env:rbm-examples-ros]*).

1.1.5 Pre-release version

The latest pre-release can be installed from the GitHub master branch:

```
$ pip install git+https://github.com/phausamann/rigid-body-motion.git
```

You can `download` this guide as a Jupyter notebook.

1.2 Reference frames

rigid_body_motion provides a flexible high-performance framework for working offline with motion data. The core of this framework is a mechanism for constructing trees of both static and dynamic reference frames that supports automatic lookup and application of transformations across the tree.

Note

The following examples require the `matplotlib` library.

---

[1]:
```
import numpy as np
import rigid_body_motion as rbm
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (6, 6)
```

1.2.1 Static frames

We will begin by defining a world reference frame using the *ReferenceFrame* class:

[2]:
```
rf_world = rbm.ReferenceFrame("world")
```

Now we can add a second reference frame as a child of the world frame. This frame is translated by 5 meters in the x direction and rotated 90° around the z axis. Note that rotations are represented as unit quaternions by default.

[3]:
```
rf_observer = rbm.ReferenceFrame(
    "observer",
    parent=rf_world,
    translation=(5, 0, 0),
    rotation=(np.sqrt(2) / 2, 0, 0, np.sqrt(2) / 2),
)
```

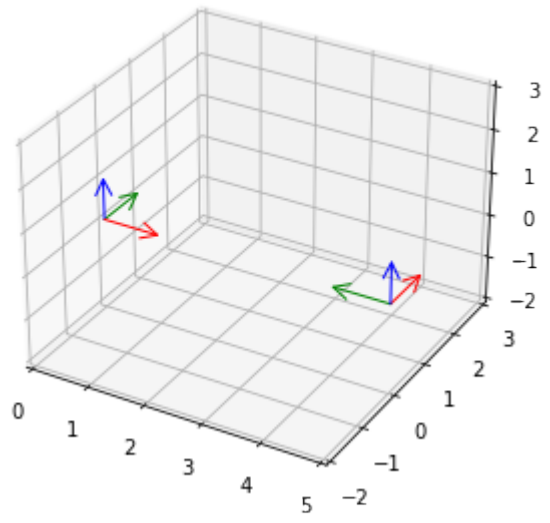
We can show the reference frame tree with the *render_tree* function:

[4]:
```
rbm.render_tree(rf_world)
```

```
world
└── observer
```

It is also possible to show a 3d plot of static reference frames with *plot.reference_frame()*:

[5]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame(rf_world, ax=ax)
rbm.plot.reference_frame(rf_observer, rf_world, ax=ax)

fig.tight_layout()
```



To facilitate referring to previously defined frames, the library has a registry where frames can be stored by name with *ReferenceFrame.register()*:

[6]:
```
rf_world.register()
rf_observer.register()
rbm.registry
```

[6]:
```
{'world': <ReferenceFrame 'world'>, 'observer': <ReferenceFrame 'observer'>}
```

1.2.2 Transformations

Now that we've set up a basic tree, we can use it to transform motion between reference frames. We use the *lookup_transform()* method to obtain the transformation from the world to the observer frame:

[7]:
```
t, r = rbm.lookup_transform(outof="world", into="observer")
```

This transformation consists of a translation t:

[8]:
```
t
```

[8]:
```
array([0., 5., 0.])
```

and a rotation r:

[9]:
```
r
```

[9]:
```
array([ 0.70710678,  0.        ,  0.        , -0.70710678])
```

Position

rigid_body_motion uses the convention that a transformation is a rotation followed by a translation. Here, when applying the transformation to a point p expressed with respect to (wrt) the world frame W it yields the point wrt the observer frame O:

$$p_O = \mathrm{rot}\left(r, p_W\right) + t$$

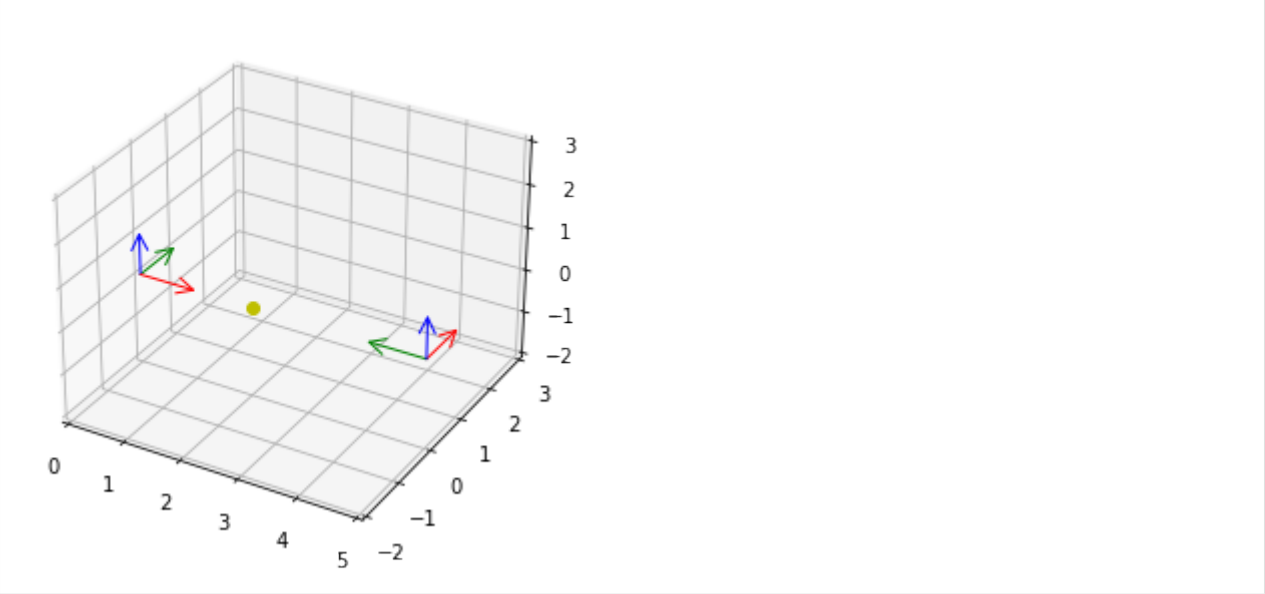
The $\mathrm{rot}()$ function denotes the rotation of a vector by a quaternion.

Let's assume we have a rigid body located at 2 meters in the x direction from the origin of the world reference frame:

[10]:
```
p_body_world = np.array((2, 0, 0))
```

We can add the body position to the plot with *plot.points()*:

[11]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame(rf_world, ax=ax)
rbm.plot.reference_frame(rf_observer, rf_world, ax=ax)
rbm.plot.points(p_body_world, ax=ax, fmt="yo")

fig.tight_layout()
```



We can use the above formula to transform the position of the body into the observer frame. The *rotate_vectors()* method implements the rotation of a vector by a quaternion:

[12]:
```
p_body_observer = rbm.rotate_vectors(r, p_body_world) + t
p_body_observer
```

[12]:
```
array([0., 3., 0.])
```

As expected, the resulting position of the body is 3 meters from the observer in the y direction. For convenience, the *transform_points()* method performs all of the above steps:

1. Lookup of the frames by name in the registry (if applicable)
2. Computing the transformation from the source to the target frame
3. Applying the transformation to the point(s)

[13]:
```
p_body_observer = rbm.transform_points(p_body_world, outof="world", into="observer")
p_body_observer
```

[13]:
```
array([0., 3., 0.])
```

Orientation

Orientations expressed in quaternions are transformed by quaternion multiplication:

$$o_O = r \cdot o_W$$

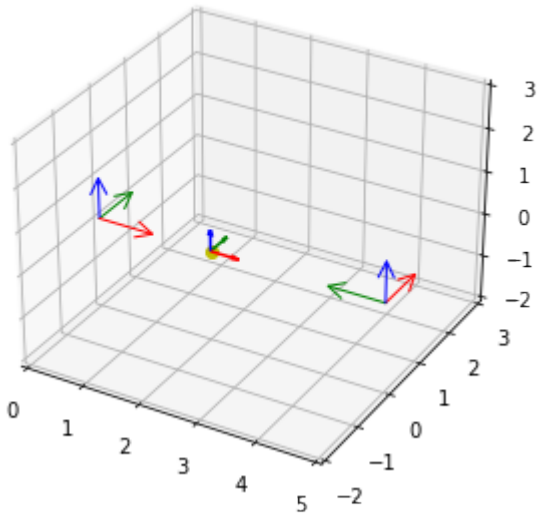
This multiplication is implemented in the *qmul()* function to which you can pass an arbtrary number of quaternions to multiply. Assuming the body is oriented in the same direction as the world frame, transforming the orientation into the observer frame results in a rotation around the yaw axis:

[14]:
```
o_body_world = np.array((1, 0, 0, 0))
rbm.qmul(r, o_body_world)
```

[14]:
```
array([ 0.70710678,  0.        ,  0.        , -0.70710678])
```

We can add the orientation to the plot with *plot.quaternions()*:

[15]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame(rf_world, ax=ax)
rbm.plot.reference_frame(rf_observer, rf_world, ax=ax)
rbm.plot.points(p_body_world, ax=ax, fmt="yo")
rbm.plot.quaternions(o_body_world, base=p_body_world, ax=ax)

fig.tight_layout()
```



Again, for convenience, the *transform_quaternions()* function can be used in the same way as *transform_points()*:

[16]:
```
rbm.transform_quaternions(o_body_world, outof="world", into="observer")
```

[16]:
```
array([ 0.70710678,  0.        ,  0.        , -0.70710678])
```

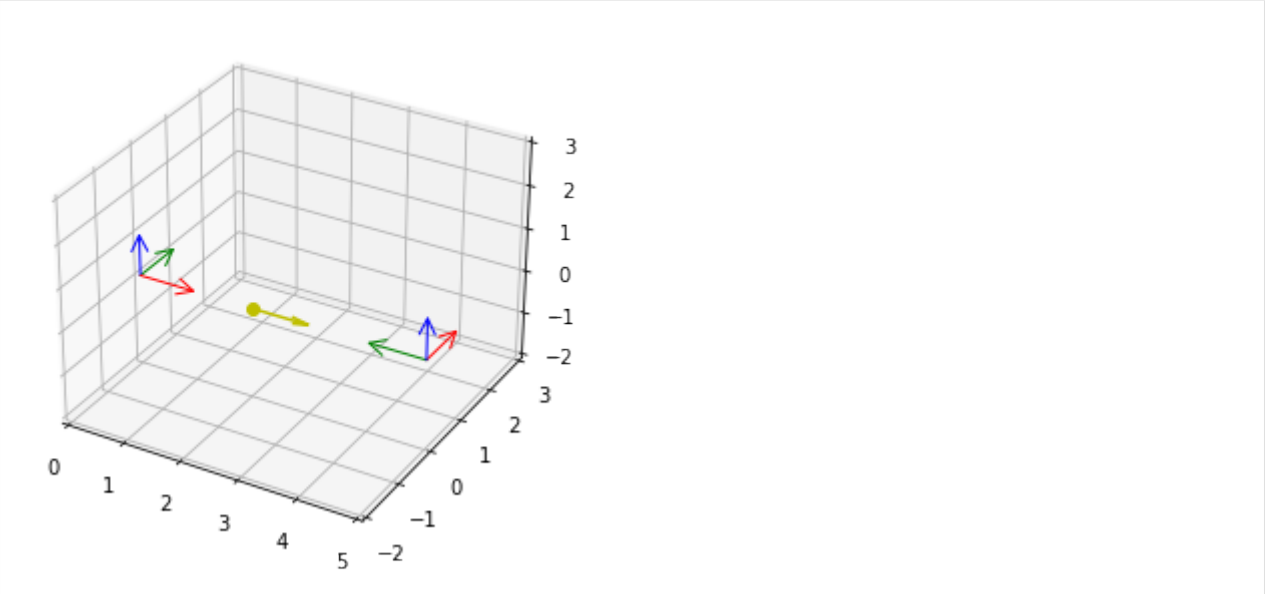
Vectors

Let's assume the body moves in the x direction with a velocity of 1 m/s:

[17]:
```
v_body_world = np.array((1, 0, 0))
```

We can add the velocity to the plot with *plot.vectors()*:

[18]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame(rf_world, ax=ax)
rbm.plot.reference_frame(rf_observer, rf_world, ax=ax)
rbm.plot.points(p_body_world, ax=ax, fmt="yo")
rbm.plot.vectors(v_body_world, base=p_body_world, ax=ax, color="y")

fig.tight_layout()
```



From the point of view of the observer, the body moves with the same speed, but in the negative y direction. Therefore, we need to apply a coordinate transformation to represent the velocity vector in the observer frame:

$$v_O = \mathrm{rot}\left(r, v_W\right)$$

[19]:
```
rbm.rotate_vectors(r, v_body_world)
```

[19]:
```
array([ 0., -1.,  0.])
```

Like before, the *transform_vectors()* function can be used in the same way as *transform_points()*:

[20]:
```
rbm.transform_vectors(v_body_world, outof="world", into="observer")
```

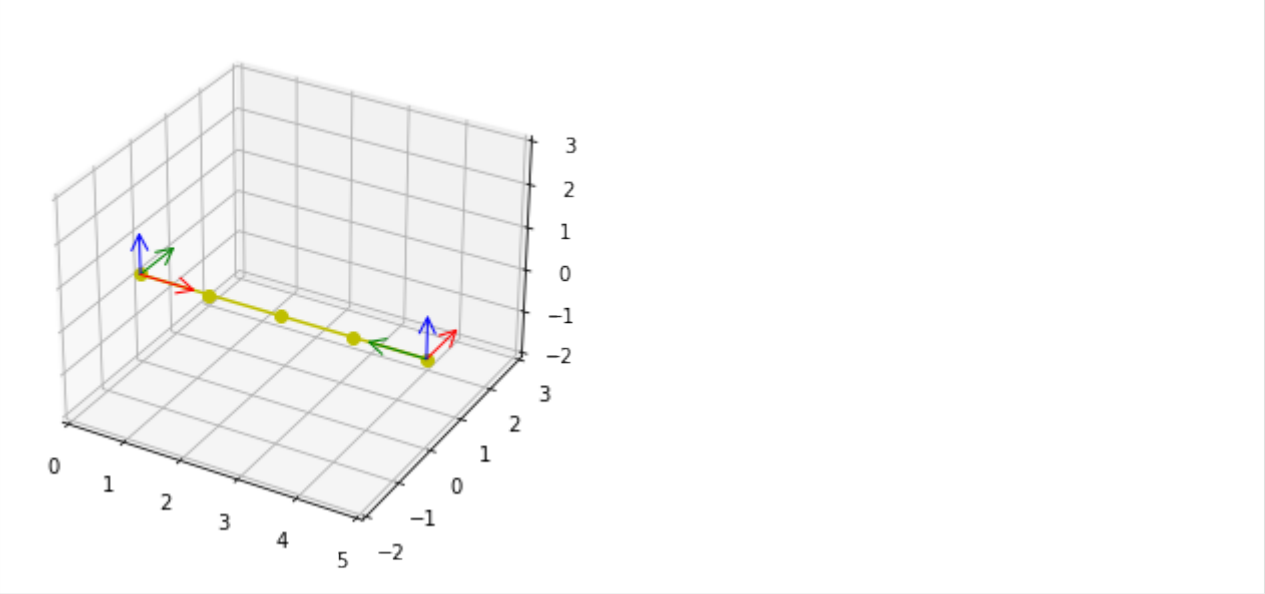
[20]:
```
array([ 0., -1.,  0.])
```

1.2.3 Moving frames

Now, let's assume that the body moves from the origin of the world frame to the origin of the observer frame in 5 steps:

[21]:
```
p_body_world = np.zeros((5, 3))
p_body_world[:, 0] = np.linspace(0, 5, 5)
```

[22]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame(rf_world, ax=ax)
rbm.plot.reference_frame(rf_observer, rf_world, ax=ax)
rbm.plot.points(p_body_world, ax=ax, fmt="yo-")

fig.tight_layout()
```



We will now attach a reference frame to the moving body to explain the handling of moving reference frames. For this, we need to associate the positions of the body with corresponding timestamps:

[23]:
```
ts_body = np.arange(5)
```

Let's construct the moving body frame and add it to the registry. We will use the *register_frame()* convenience method:

[24]:
```
rbm.register_frame("body", translation=p_body_world, timestamps=ts_body, parent="world")
rbm.render_tree("world")
```

```
world
├── observer
└── body
```

If we transform a static point from the world into the body frame its position will change over time, which is why *transform_points()* will return an array of points even though we pass only a single point:

[25]:
```
rbm.transform_points((2, 0, 0), outof="world", into="body")
```

[25]:
```
array([[ 2.  ,  0.  ,  0.  ],
       [ 0.75,  0.  ,  0.  ],
       [-0.5 ,  0.  ,  0.  ],
       [-1.75,  0.  ,  0.  ],
       [-3.  ,  0.  ,  0.  ]])
```

One of the central features of the reference frame mechanism is its ability to consolidate arrays of timestamped motion even when the timestamps don't match. To illustrate this, let's create a second body moving in the y direction in world coordinates whose timestamps are offset by 0.5 seconds compared to the first body:

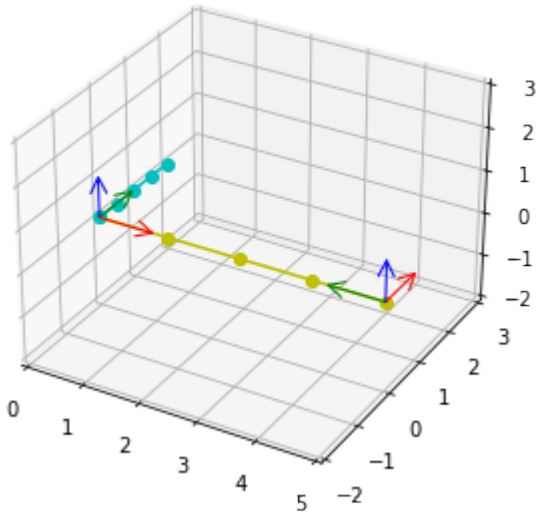
[26]:
```
p_body2_world = np.zeros((5, 3))
p_body2_world[:, 1] = np.linspace(0, 2, 5)
ts_body2 = ts_body - 0.5
```

[27]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
```

(continues on next page)

(continued from previous page)

```
rbm.plot.reference_frame(rf_world, ax=ax)
rbm.plot.reference_frame(rf_observer, rf_world, ax=ax)
rbm.plot.points(p_body_world, ax=ax, fmt="yo-")
rbm.plot.points(p_body2_world, ax=ax, fmt="co-")

fig.tight_layout()
```



Transforming the position of the second body into the frame of the first body still works, despite the timestamp mismatch:

[28]:
```
p_body2_body, ts_body2_body = rbm.transform_points(
    p_body2_world,
    outof="world",
    into="body",
    timestamps=ts_body2,
    return_timestamps=True,
)
```

This is because behind the scenes, *transform_points()* matches the timestamps of the array to transform with those of the transformation across the tree by

1. computing the range of timestamps for which the transformation is defined,
2. intersecting that range with the range of timestamps to be transformed and
3. interpolating the resulting transformation across the tree to match the timestamps of the array.

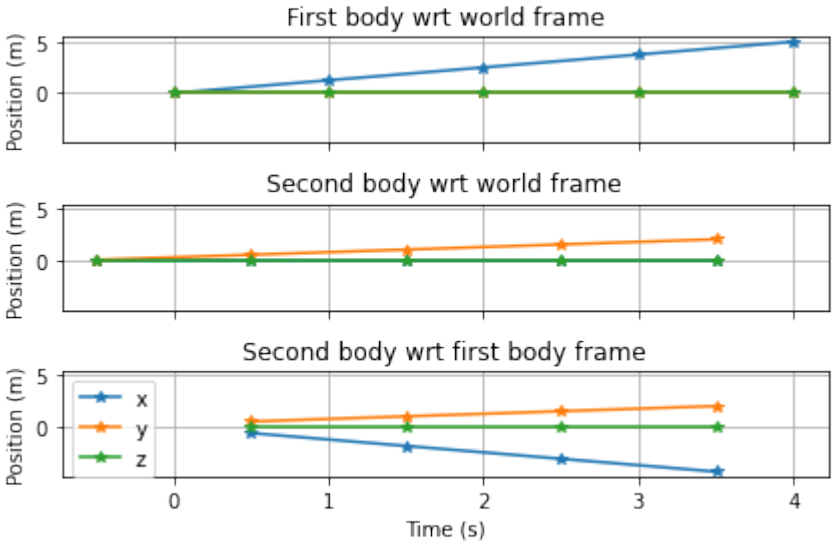
Note that we specified `return_timestamps=True` to obtain the timestamps of the transformed array as they are different from the original timestamps. Let's plot the position of both bodies wrt the world frame as well as the position of the second body wrt the first body to see how the timestamp matching works:

[29]:
```
fig, axarr = plt.subplots(3, 1, sharex=True, sharey=True)

axarr[0].plot(ts_body, p_body_world, "*-")
axarr[0].set_ylabel("Position (m)")
```

(continues on next page)

(continued from previous page)

```
axarr[0].set_title("First body wrt world frame")
axarr[0].grid("on")

axarr[1].plot(ts_body2, p_body2_world, "*-")
axarr[1].set_ylabel("Position (m)")
axarr[1].set_title("Second body wrt world frame")
axarr[1].grid("on")

axarr[2].plot(ts_body2_body, p_body2_body, "*-")
axarr[2].set_xlabel("Time (s)")
axarr[2].set_ylabel("Position (m)")
axarr[2].set_title("Second body wrt first body frame")
axarr[2].grid("on")
axarr[2].legend(["x", "y", "z"], loc="upper left")

fig.tight_layout()
```



As you can see, the resulting timestamps are the same as those of the second body; however, the first sample has been dropped because the transformation is not defined there.

You can `download` this guide as a Jupyter notebook.

1.3 Linear and angular velocity

We have seen how to look up and transform positions and orientations across reference frames in the previous section. Working with velocities adds some complexity that will be explained in this section.

Note

The following examples require the `matplotlib` library.

---

[1]:
```
import numpy as np
import rigid_body_motion as rbm
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (6, 6)
```

1.3.1 Setting up

Like in the previous section, we first set up the world frame:

[2]:
```
rbm.register_frame("world")
```

For simplicity, all reference frames will share the same timestamps:

[3]:
```
n_timestamps = 5
ts = np.linspace(0, 5, n_timestamps)
```

We define a body moving 5 meters in the x direction:

[4]:
```
p_body_world = np.zeros((n_timestamps, 3))
p_body_world[:, 0] = np.linspace(0, 5, n_timestamps)
```

In addition, this body will rotate 90° around the z axis. We use helper functions from the `quaternion` package for this:

[5]:
```
from quaternion import as_float_array, from_euler_angles

o_body_world = as_float_array(
    from_euler_angles(np.linspace(0, np.pi / 4, n_timestamps), 0, 0)
)
```

Now we can attach a reference frame to this body:

[6]:
```
rbm.register_frame(
    "body",
    translation=p_body_world,
    rotation=o_body_world,
    timestamps=ts,
    parent="world",
)
```

We now define a second moving body whose motion we describe wrt the frame of the first body. It is located at 1 meter in the y direction and moves 1 meter in the negative x direction.

[7]:
```
p_body2_body = np.zeros((n_timestamps, 3))
p_body2_body[:, 0] = -np.linspace(0, 1, n_timestamps)
p_body2_body[:, 1] = 1
```

This body also rotates, but this time around the y axis:

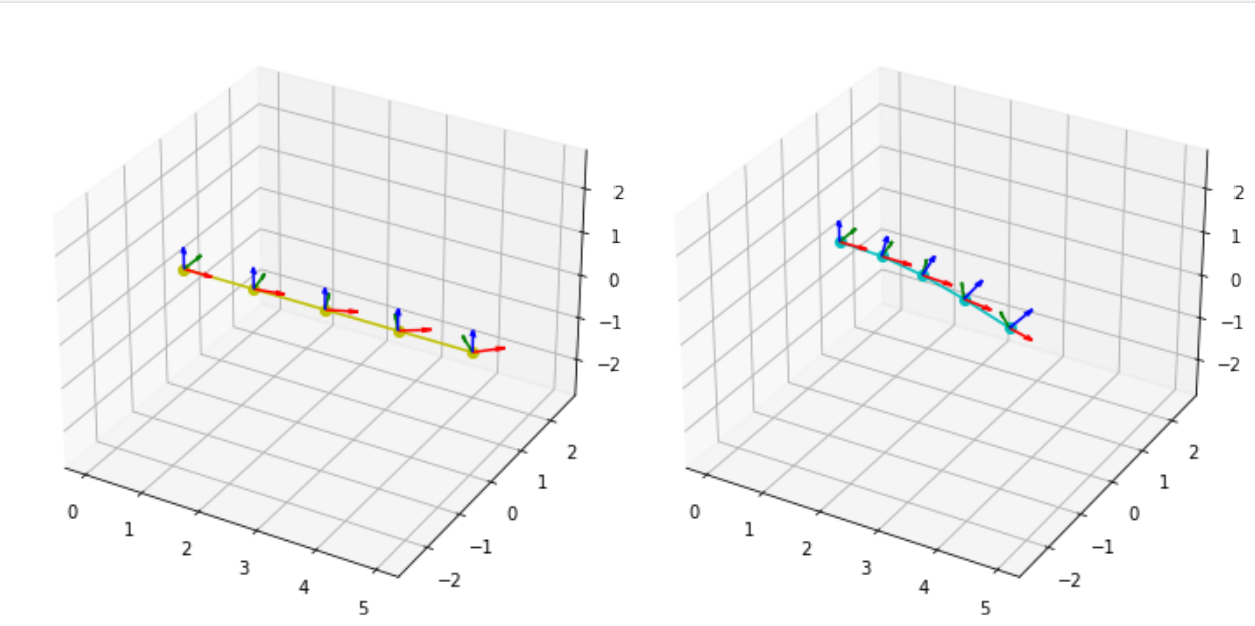
[8]:
```
o_body2_body = as_float_array(
    from_euler_angles(0, np.linspace(0, np.pi / 4, n_timestamps), 0)
)
```

Now we can register a frame attached to the second body as a child frame of the first body frame:

[9]:
```
rbm.register_frame(
    "body2",
    translation=p_body2_body,
    rotation=o_body2_body,
    timestamps=ts,
    parent="body",
)
```

Let's plot the position and orientation of both bodies wrt the world frame. We use the *lookup_pose()* method to obtain the position of the second body in the world frame:

[10]:
```
p_body2_world, o_body2_world = rbm.lookup_pose("body2", "world")

fig = plt.figure(figsize=(10, 5))

ax = fig.add_subplot(121, projection="3d")
rbm.plot.points(p_body_world, ax=ax, fmt="yo-")
rbm.plot.quaternions(o_body_world, base=p_body_world, ax=ax)

ax = fig.add_subplot(122, projection="3d", sharex=ax, sharey=ax, sharez=ax)
rbm.plot.points(p_body2_world, ax=ax, fmt="co-")
rbm.plot.quaternions(o_body2_world, base=p_body2_world, ax=ax)

fig.tight_layout()
```



1.3.2 Linear velocity

The linear velocity of a frame wrt another frame can be calculated with the *lookup_linear_velocity()* method:

[11]:
```
v_body2_body = rbm.lookup_linear_velocity("body2", "body")
v_body2_body[0]
```

[11]:
```
array([-0.2,  0. ,  0. ])
```

As expected, the velocity of the second body wrt the first body $v_{B2/B}$ is 0.2 m/s in the negative x direction. Next, we will see how we can obtain the velocity of the second body wrt the world frame W.

Transforming the reference frame

When transforming linear velocities across frames, we need to use the so-called "three term velocity formula". In this case, we have the velocity of the *moving frame* $B2$ wrt the *reference frame* B. We can transform the current reference frame B to the new frame W as follows:

$$v_{B2/W} = \underbrace{v_{B2/B}}_{\text{Input}} + \underbrace{v_{B/W} + \omega_{B/W} \times t_{B2/B}}_{\text{Lookup}}$$

In addition to the velocities between the reference frames B and W, this formula also requires the translation between the moving frame $B2$ and the original reference frame B. This is why we also need to specify the `moving_frame` argument when using *transform_linear_velocity()*:

[12]:
```
v_body2_world = rbm.transform_linear_velocity(
    v_body2_body, outof="body", into="world", moving_frame="body2", timestamps=ts,
)
```

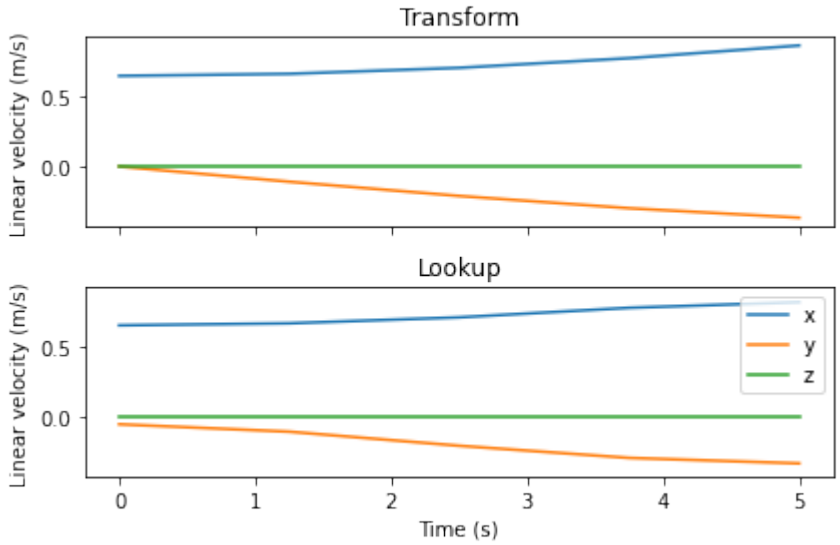
Alternatively, we can also use *lookup_linear_velocity()* to lookup the position of $B2$ wrt W and differentiate:

[13]:
```
v_body2_world_lookup = rbm.lookup_linear_velocity("body2", "world")
```

The following short helper function can be used to compare the two methods:

[14]:
```
def compare_velocities(transform, lookup, timestamps=None, mode="linear"):
    """ Compare velocities from transform and lookup. """
    fig, axarr = plt.subplots(2, 1, sharex=True, sharey=True)

    ylabel = f"{mode.capitalize()} velocity ({'rad/s' if mode == 'angular' else 'm/s'})"

    axarr[0].plot(timestamps, transform)
    axarr[0].set_ylabel(ylabel)
    axarr[0].set_title("Transform")

    axarr[1].plot(timestamps, lookup)
    axarr[1].set_xlabel("Time (s)")
    axarr[1].set_ylabel(ylabel)
    axarr[1].set_title("Lookup")
    axarr[1].legend(["x", "y", "z"])

    fig.tight_layout()
```

[15]:
```
compare_velocities(v_body2_world, v_body2_world_lookup, ts)
```



We see a slight discrepancy due to numerical inconsistencies in the derivative calculation. However, these errors are reduced at higher sampling rates.

Transforming the moving frame

In a different scenario, we might be given the velocity of the first body wrt the world frame and want to transform the *moving frame* from B to $B2$ to compute the velocity of the second body wrt W. The same formula applies, although this time the input is $v_{B/W}$:

$$v_{B2/W} = \underbrace{v_{B/W}}_{\text{Input}} + \underbrace{v_{B2/B} + \omega_{B/W} \times t_{B2/B}}_{\text{Lookup}}$$

When using *transform_linear_velocity()* we need to be careful that the velocity is represented in the coordinates of the frame we want to transform. Therefore, $v_{B/W}$ has to be represented in B:

[16]:
```
v_body_world = rbm.lookup_linear_velocity("body", "world", represent_in="body")
```

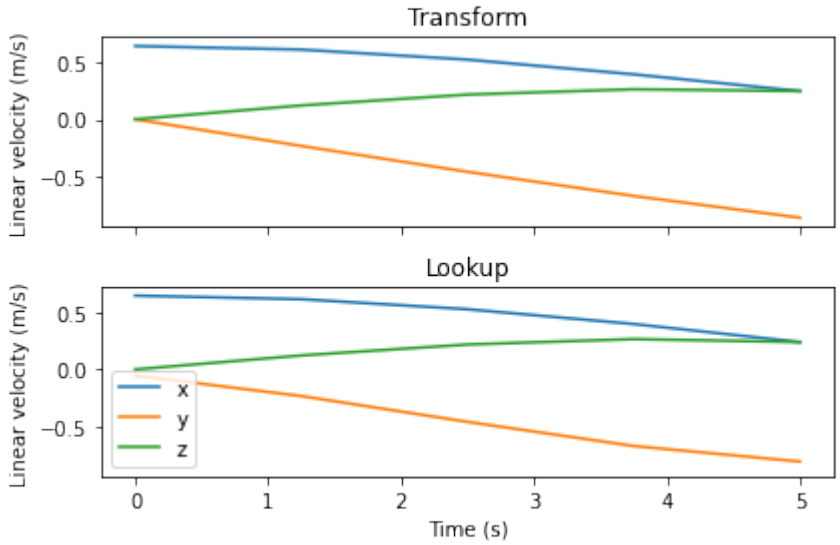
Now we can use *transform_linear_velocity()* with `what="moving_frame"` to transform from B to $B2$. Since the method needs to look up $\omega_{B/W}$, this time we need to provide the `reference_frame` of the velocity:

[17]:
```
v_body2_world = rbm.transform_linear_velocity(
    v_body_world,
    outof="body",
    into="body2",
    reference_frame="world",
    what="moving_frame",
    timestamps=ts,
)
```

Let's compare this result against the lookup method again. Note that the transformation also changes the representation frame to the new frame, which is why the resulting velocity is different compared to the first case:

[18]:
```
v_body2_world_lookup = rbm.lookup_linear_velocity("body2", "world", represent_in="body2")
```

[19]:
```
compare_velocities(v_body2_world, v_body2_world_lookup, ts)
```



1.3.3 Angular velocity

Angular velocities can be looked up with *lookup_angular_velocity()*:

[20]:
```
w_body2_body = rbm.lookup_angular_velocity("body2", "body")
w_body2_body[0]
```

[20]:
```
array([0.        , 0.15708158, 0.        ])
```

Transforming the reference frame

Transforming the reference frame of angular velocity is similar to the case of linear velocity, although the formula is a lot simpler:

$$\omega_{B2/W} = \underbrace{\omega_{B2/B}}_{\text{Input}} + \underbrace{\omega_{B/W}}_{\text{Lookup}}$$

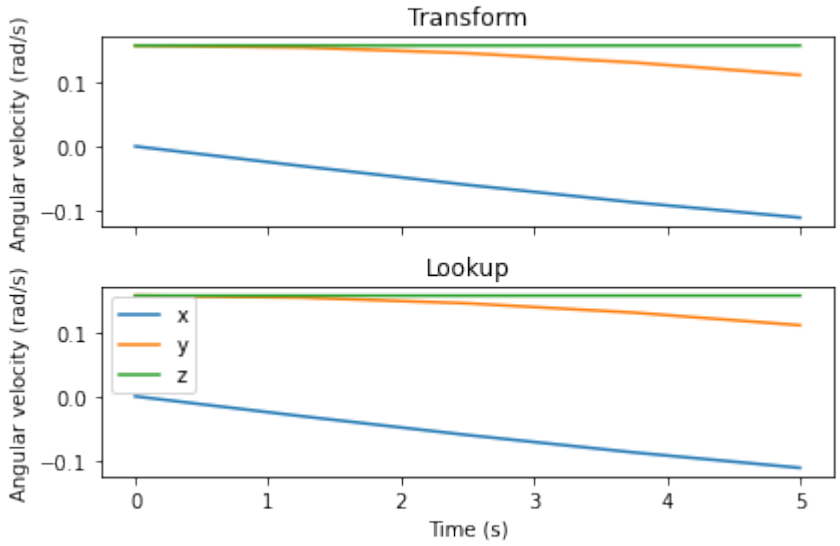
Because of this, *transform_angular_velocity()* also has a simpler interface:

[21]:
```
w_body2_world = rbm.transform_angular_velocity(
    w_body2_body, outof="body", into="world", timestamps=ts
)
```

Comparing the transform to the lookup shows no differences in the result:

[22]:
```
w_body2_world_lookup = rbm.lookup_angular_velocity("body2", "world")
```

[23]:
```
compare_velocities(w_body2_world, w_body2_world_lookup, ts, mode="angular")
```



Transforming the moving frame

As before, we can also transform the moving frame:

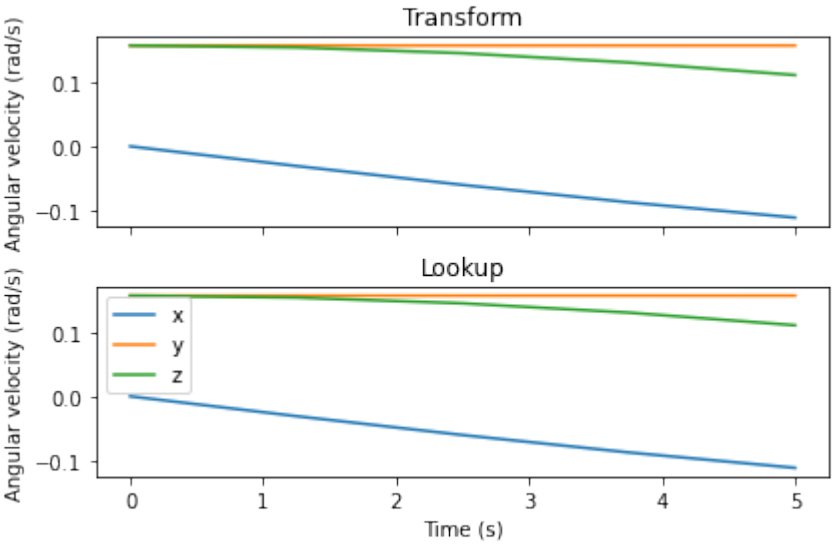
$$\omega_{B2/W} = \underbrace{\omega_{B/W}}_{\text{Input}} + \underbrace{\omega_{B2/B}}_{\text{Lookup}}$$

[24]:
```
w_body_world = rbm.lookup_angular_velocity("body", "world", represent_in="body")
```

[25]:
```
w_body2_world = rbm.transform_angular_velocity(
    w_body_world, outof="body", into="body2", what="moving_frame", timestamps=ts,
)
```

[26]:
```
w_body2_world_lookup = rbm.lookup_angular_velocity("body2", "world", represent_in="body2
↪")
```

[27]:
```
compare_velocities(w_body2_world, w_body2_world_lookup, ts, mode="angular")
```



You can `download` this guide as a Jupyter notebook.

1.4 Estimating transforms from data

It is often necessary to estimate transformations between rigid bodies that are not explicitly known. This happens for example when the motion of the same rigid body is measured by different tracking systems that represent their data in different world frames.

Note

The following examples require the `matplotlib` library.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import rigid_body_motion as rbm

plt.rcParams["figure.figsize"] = (10, 5)
```

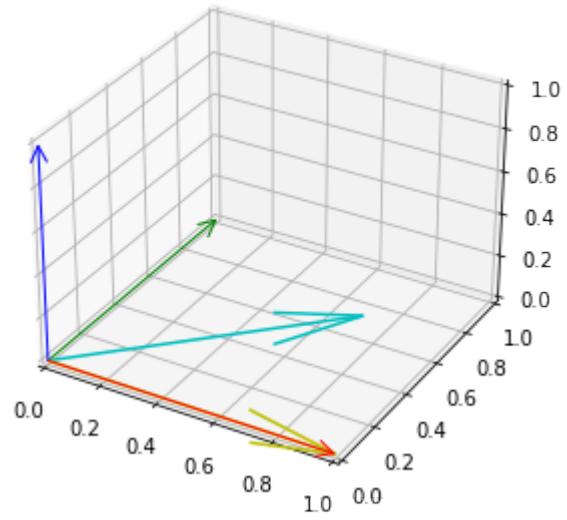
[2]:
```
rbm.register_frame("world")
```

1.4.1 Shortest arc rotation

Let's assume we have two vectors v_1 and v_2:

[3]:
```
v1 = (1, 0, 0)
v2 = (np.sqrt(2) / 2, np.sqrt(2) / 2, 0)
```

[4]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame("world", ax=ax)
rbm.plot.vectors(v1, ax=ax, color="y")
rbm.plot.vectors(v2, ax=ax, color="c")

fig.tight_layout()
```



The quaternion r that rotates v_1 in the same direction as v_2, i.e., that satisfies:

$$\frac{v_2}{\|v_2\|} = \frac{\mathrm{rot}(r, v_1)}{\|v_1\|}$$

can be computed with the *shortest_arc_rotation()* method:

[5]:
```
rbm.shortest_arc_rotation(v1, v2)
```

[5]:
```
array([0.92387953, 0.        , 0.        , 0.38268343])
```

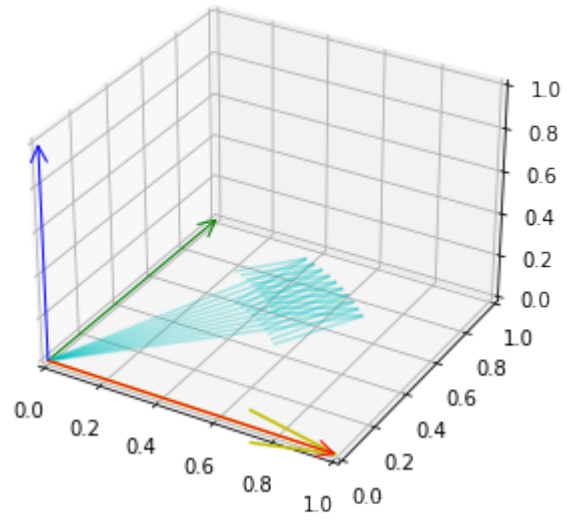
The method also works with arrays of vectors. Let's first construct an array of progressive rotations around the yaw axis with the *from_euler_angles()* method:

[6]:
```
r = rbm.from_euler_angles(yaw=np.linspace(0, np.pi / 8, 10))
```

Now we can rotate v_2 with r. Because we rotate a single vector with multiple quaternions we have to specify `one_to_one=False`:

[7]:
```
v2_arr = rbm.rotate_vectors(r, v2, one_to_one=False)
```

[8]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame("world", ax=ax)
rbm.plot.vectors(v1, ax=ax, color="y")
rbm.plot.vectors(v2_arr, ax=ax, color="c", alpha=0.3)

fig.tight_layout()
```



shortest_arc_rotation() now returns an array of quaternions:

[9]:
```
rbm.shortest_arc_rotation(v1, v2_arr)
```

[9]:
```
array([[0.92387953, 0.        , 0.        , 0.38268343],
       [0.91531148, 0.        , 0.        , 0.40274669],
       [0.90630779, 0.        , 0.        , 0.42261826],
       [0.89687274, 0.        , 0.        , 0.44228869],
       [0.88701083, 0.        , 0.        , 0.46174861],
       [0.87672676, 0.        , 0.        , 0.48098877],
       [0.8660254 , 0.        , 0.        , 0.5       ],
       [0.85491187, 0.        , 0.        , 0.51877326],
       [0.84339145, 0.        , 0.        , 0.53729961],
       [0.83146961, 0.        , 0.        , 0.55557023]])
```

1.4.2 Best fit rotation

In a different scenario, we might have two vectors that are offset by a fixed rotation and are rotating in space:

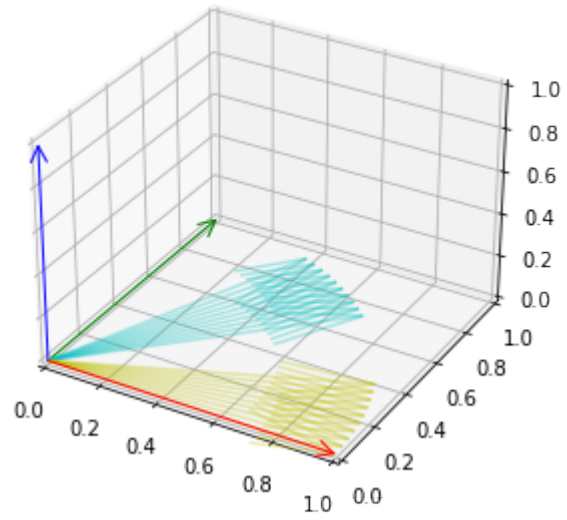
[10]:
```
v1_arr = rbm.rotate_vectors(r, v1, one_to_one=False)
```

[11]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
```

(continues on next page)

(continued from previous page)

```
rbm.plot.reference_frame("world", ax=ax)
rbm.plot.vectors(v1_arr, ax=ax, color="y", alpha=0.3)
rbm.plot.vectors(v2_arr, ax=ax, color="c", alpha=0.3)

fig.tight_layout()
```



The rotation between the vectors can be found with a least-squares minimization:

$$\min_r \|v_2 - \mathrm{rot}(r, v_1)\|$$

This is implemented in the *best_fit_rotation()* method:

[12]:
```
rbm.best_fit_rotation(v1_arr, v2_arr)
```

[12]:
```
array([ 0.92387953, -0.        , -0.        ,  0.38268343])
```

1.4.3 Best fit transform

In yet another case, we might have two arrays of points (e.g. point clouds) with a fixed transform (rotation *and* translation) between them:

[13]:
```
p1_arr = 0.1 * np.random.randn(100, 3)
```

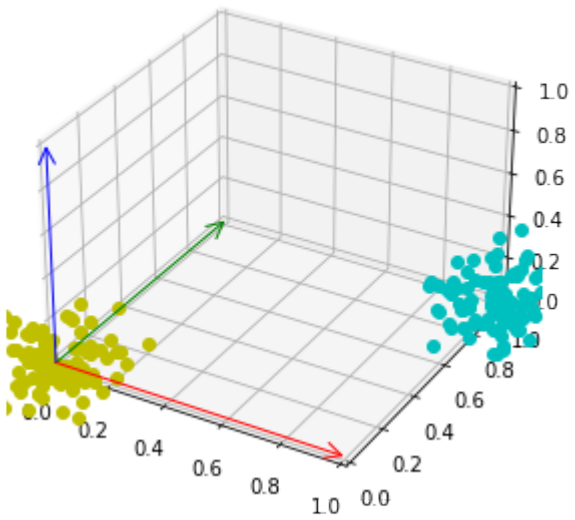
[14]:
```
t = np.array((1, 1, 0))
r = rbm.from_euler_angles(yaw=np.pi / 4)
p2_arr = rbm.rotate_vectors(r, p1_arr, one_to_one=False) + t
```

[15]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

rbm.plot.reference_frame("world", ax=ax)
rbm.plot.points(p1_arr, ax=ax, fmt="yo")
```

(continues on next page)

(continued from previous page)

```
rbm.plot.points(p2_arr, ax=ax, fmt="co")

fig.tight_layout()
```



To estimate this transform, we can minimize:

$$\min_r \|p_2 - (\text{rot}(r, p_1) + t)\|$$

This algorithm (also called point set registration) is implemented in the *best_fit_transform()* method:

[16]:
```
rbm.best_fit_transform(p1_arr, p2_arr)
```

[16]:
```
(array([ 1.00000000e+00,  1.00000000e+00, -1.51788304e-18]),
 array([9.23879533e-01, 1.21634372e-16, 5.55111512e-16, 3.82683432e-01]))
```

1.4.4 Iterative closest point

The above algorithm only works for known correspondences between points p_1 and p_2 (i.e., each point in `p1_arr` corresponds to the same index in `p2_arr`). This is not always the case - in fact, something like a point cloud from different laser scans of the same object might yield sets of completely different points. An approximate transform can still be found with the iterative closest point (ICP) algorithm. We can simulate the case of unknown correspondences by randomly permuting the second array:

[17]:
```
rbm.iterative_closest_point(p1_arr, np.random.permutation(p2_arr))
```

[17]:
```
(array([1.00368784, 0.88399277, 0.00839297]),
 array([-0.87954944,  0.15187444,  0.42940227, -0.13762494]))
```

Note that there is a discrepancy in the estimated transform compared to the best fit transform. ICP usually yields better results with a larger number of points that have more spatial structure.

You can `download` this guide as a Jupyter notebook.

1.5 Working with xarray

rigid_body_motion provides first class support for xarray data types. xarray has several features that make working with motion data convenient:

1. xarray is designed to combine physical data with metadata such as timestamps.
2. xarray's `Dataset` class can be used as a container for timestamped transformations.
3. Arbitrary metadata can be attached to arrays to keep track of e.g. reference frames.

We recommend you to familiarize yourself with xarray before working through this tutorial. Their documentation is an excellent resource for that.

Note

The following examples require the `matplotlib`, `xarray` and `netcdf4` libraries.

---

[1]:
```
import rigid_body_motion as rbm
import xarray as xr
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (10, 5)
```

1.5.1 Loading example data

rigid_body_motion includes a recording of head and eye tracking data (using the Intel RealSense T265 as the head tracker and the Pupil Core eye tracker). This data can be loaded with `xr.open_dataset`:

[2]:
```
head = xr.open_dataset(rbm.example_data["head"])
head
```

[2]:
```
<xarray.Dataset>
Dimensions:           (time: 66629, cartesian_axis: 3, quaternion_axis: 4)
Coordinates:
  * time              (time) datetime64[ns] 2020-02-02T00:27:14.300365210 ...
  * cartesian_axis    (cartesian_axis) object 'x' 'y' 'z'
  * quaternion_axis   (quaternion_axis) object 'w' 'x' 'y' 'z'
Data variables:
    position          (time, cartesian_axis) float64 ...
    linear_velocity   (time, cartesian_axis) float64 ...
    angular_velocity  (time, cartesian_axis) float64 ...
    confidence        (time) float64 ...
    orientation       (time, quaternion_axis) float64 ...
```

The dataset includes position and orientation as well as angular and linear velocity of the tracker. Additionally, it includes the physical dimensions `time`, `cartesian_axis` (for position and velocities) and `quaternion_axis` (for orientation). Let's have a look at the position data:

[3]:
```
head.position
```

[3]:
```
<xarray.DataArray 'position' (time: 66629, cartesian_axis: 3)>
[199887 values with dtype=float64]
Coordinates:
```

(continues on next page)

(continued from previous page)

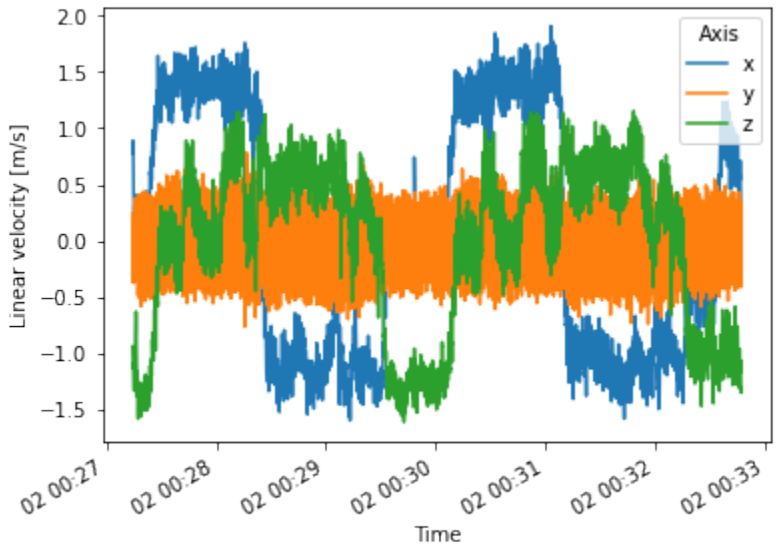
```
  * time            (time) datetime64[ns] 2020-02-02T00:27:14.300365210 ... 2...
  * cartesian_axis  (cartesian_axis) object 'x' 'y' 'z'
Attributes:
    long_name:  Position
    units:      m
```

As you can see, this is a two-dimensional array (called `DataArray` in xarray) with timestamps and explicit names for the physical axes in cartesian coordinates.

xarray also provides a straightforward way of plotting:

[4]:
```
head.linear_velocity.plot.line(x="time")
```

[4]:
```
[<matplotlib.lines.Line2D at 0x7f56399ba1d0>,
 <matplotlib.lines.Line2D at 0x7f56399e2650>,
 <matplotlib.lines.Line2D at 0x7f56399e27d0>]
```



The example recording is from a test subject wearing the combined head/eye tracker while walking twice around a building. The head tracking data is represented in a world-fixed reference frame whose origin is at the head tracker's location at the start of the recording.

In the next step, we will leverage rigid_body_motion's powerful reference frame mechanism to transform the linear velocity from world to tracker coordinates.

1.5.2 Reference frame interface

As in the previous tutorial, we begin by registering the world frame as root of the reference frame tree:

[5]:
```
rbm.register_frame("world")
```

Timestamped reference frames can be easily constructed from `Dataset` instances with the *ReferenceFrame.from_dataset()* method. We need to specify the variables representing translation and rotation of the reference frame as well as the name of the coordinate containing timestamps and the parent frame:

[6]:
```
rf_head = rbm.ReferenceFrame.from_dataset(
    head, "position", "orientation", "time", parent="world", name="head"
)
```

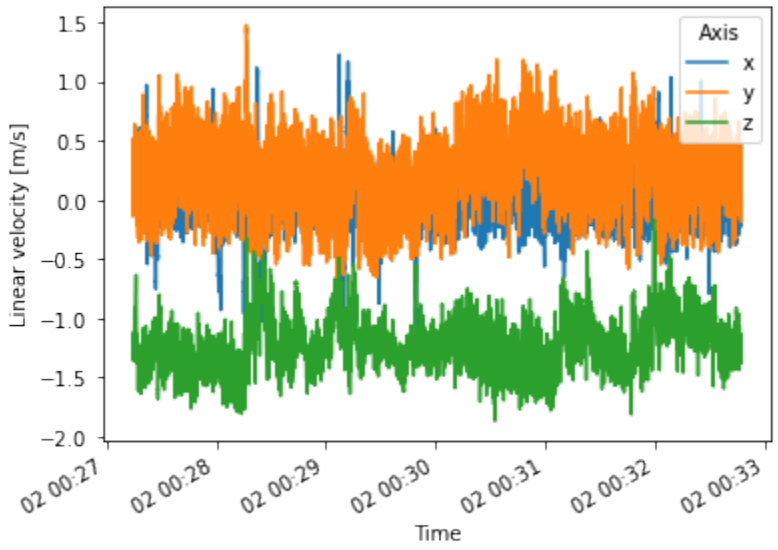
Let's register this reference frame so that we can use it easily for transformations:

[7]:
```
rf_head.register()
```

Now we can use *transform_linear_velocity()* to transform the linear velocity to be represented in tracker coordinates:

[8]:
```
v_head = rbm.transform_linear_velocity(
    head.linear_velocity, outof="world", into="head", what="representation_frame"
)
v_head.plot.line(x="time")
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7f563989bd10>,
 <matplotlib.lines.Line2D at 0x7f563991c710>,
 <matplotlib.lines.Line2D at 0x7f56398cfb50>]
```



We now see a mean linear velocity of ~1.4 m/s in the negative z direction. This is due to the coordinate system defined by the RealSense T265 where the positive z direction is defined towards the back of the device.

You can `download` this guide as a Jupyter notebook.

1.6 ROS integration

rigid_body_motion supports certain ROS functionality, provided the Python environment has been set up with the required ROS packages. This guide assumes that you are at least somewhat familiar with ROS concepts such as nodes, publishers/subscribers and messages. If not, the ROS tutorials are a good place to start.

You also need to set up a couple of dependencies which can be done very conveniently if you are using an Anaconda Python distribution. See *the ROS dependencies installation guide* for further information.

Note

The following examples require the `pooch`, `xarray`, `netcdf4` and `ipywidgets` libraries.

[1]:
```
import numpy as np
import pandas as pd
import rigid_body_motion as rbm
```

1.6.1 Loading data from rosbag files

Data can be loaded from rosbag files into numpy arrays. So far, `geometry_msgs/TransformStamped` and `nav_msgs/Odometry` messages are supported. This is done through the *RosbagReader* class:

[2]:
```
reader = rbm.ros.RosbagReader(rbm.example_data["rosbag"])
```

The reader can be used as a context manager to facilitate opening and closing of the rosbag. The `get_topics_and_types` method returns a dict with topic names and corresponding message types:

[3]:
```
with reader:
    info = reader.get_topics_and_types()
info
```

[3]:
```
{'/pupil/left_eye/transform': 'geometry_msgs/TransformStamped',
 '/pupil/right_eye/transform': 'geometry_msgs/TransformStamped',
 '/t265/transform': 'geometry_msgs/TransformStamped'}
```

Note

If you get an `ModuleNotFoundError: No module named 'rosbag'` at this point, there is an issue with the ROS dependencies. See *the ROS dependencies troubleshooting guide* to fix this.

The data included in the example rosbag is from a head/eye tracking study and contains head-in-world pose estimated by the Intel RealSense T265 as well as eye-in-head pose for both eyes estimated by the Pupil Core eye tracker.

The `load_messages` method returns a dict with the data from a specified topic:

[4]:
```
with reader:
    head = reader.load_messages("/t265/transform")
head
```

[4]:
```
{'timestamps': array([1.58060323e+09, 1.58060323e+09, 1.58060323e+09, ...,
        1.58060357e+09, 1.58060357e+09, 1.58060357e+09]),
 'position': array([[15.9316,  0.8211, 10.5429],
        [15.9354,  0.8208, 10.5382],
        [15.9393,  0.8204, 10.5335],
        ...,
        [29.8883,  2.8952,  7.6317],
        [29.8888,  2.8943,  7.6249],
        [29.8892,  2.8935,  7.6182]]),
 'orientation': array([[-0.9687,  0.0917,  0.2306,  0.0039],
        [-0.969 ,  0.0915,  0.2295,  0.005 ],
        [-0.9693,  0.0912,  0.2285,  0.0061],
```

(continues on next page)

(continued from previous page)

```
        ...,
        [-0.9915,  0.0915,  0.0929, -0.0022],
        [-0.9914,  0.0922,  0.0927, -0.0017],
        [-0.9913,  0.0932,  0.0925, -0.001 ]])}
```

Now we can construct a reference frame tree with this data:

[5]:
```
rbm.register_frame("world")
```

The T265 uses the VR coordinate convention (x right, y up, z towards the back of the device) which differs from the default ROS convention (x forward, y left, z up):

[6]:
```
R_T265_ROS = np.array([[0.0, 0.0, -1.0], [-1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])

rbm.ReferenceFrame.from_rotation_matrix(
    R_T265_ROS, parent="world", name="t265/world"
).register()
```

The recorded data describes the transformation from the T265 world frame to the tracker frame:

[7]:
```
rbm.register_frame(
    "t265/tracker",
    parent="t265/world",
    translation=head["position"],
    rotation=head["orientation"],
    timestamps=pd.to_datetime(head["timestamps"], unit="s"),
)
```

For visualization, we define an additional `"head"` frame that represents the tracker pose with the ROS coordinate convention:

[8]:
```
rbm.ReferenceFrame.from_rotation_matrix(
    R_T265_ROS, parent="t265/tracker", name="head", inverse=True,
).register()
```

[9]:
```
rbm.render_tree("world")
```

```
world
└── t265/world
    └── t265/tracker
        └── head
```

1.6.2 Visualization with RViz

This section will show you how to broadcast reference frame transforms on the `/tf` topic as well as publish other messages that are useful for visualization in RViz. If you are not familiar with RViz and/or tf, check out the RViz user guide and the tf package documentation. You can download an `.rviz` file where all topics created in the following are already set up here.

We start by creating a node for this notebook with the *init_node()* method. This method will also automatically start a `roscore` when `start_master=True` and another ROS master isn't already running:

[10]:
```
master = rbm.ros.init_node("rbm_vis", start_master=True)
```

```
started roslaunch server http://DESKTOP:41375/
ros_comm version 1.15.9


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.9

NODES

auto-starting new master
process[master]: started with pid [15323]
ROS_MASTER_URI=http://localhost:11311
setting /run_id to master
process[rosout-1]: started with pid [15333]
started core service [/rosout]
```

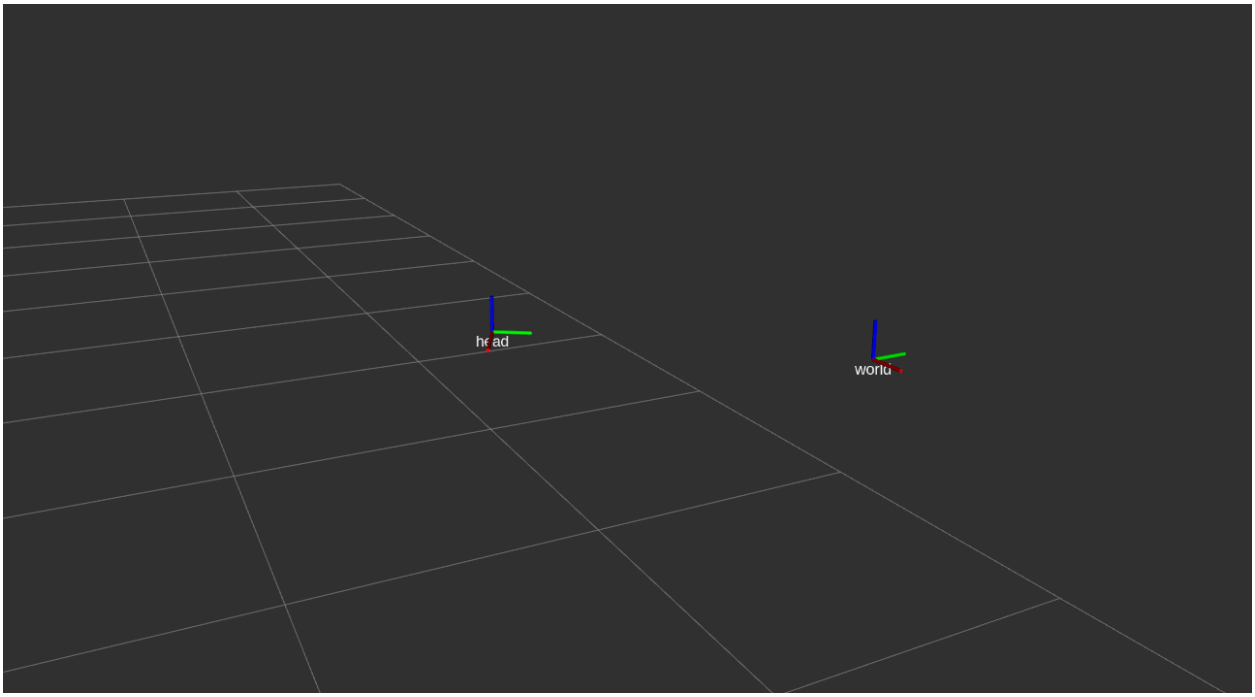
Transforms between reference frames can easily be broadcast on the `/tf` topic with the *ReferenceFrameTransformBroadcaster* class:

[11]:
```
tf_world_head = rbm.ros.ReferenceFrameTransformBroadcaster("head", base="world")
```

When calling *publish* the transform between world and head frame will be broadcast on the `/tf` topic. Since the head frame is a moving frame this will broadcast the first valid transform between the two frames by default:

[12]:
```
tf_world_head.publish()
```

RViz should now show both frames:



The broadcaster stores all valid transforms between the two frames in the `translation`, `rotation` and `timestamps` attributes:

[13]:
```
tf_world_head.translation.shape
```

[13]:
```
(66629, 3)
```

You can broadcast the transform between two frames at different points in time by specifying an index into these arrays as an argument to *publish*:

[14]:
```
tf_world_head.publish(1000)
```

Note

When "going back in time", i.e., broadcasting transforms with timestamps older than the latest broadcast timestamp, RViz will not update the tf display and you may get a `TF_OLD_DATA` warning in the console. When this happens, click on the "Reset" button in the lower left corner in RViz.

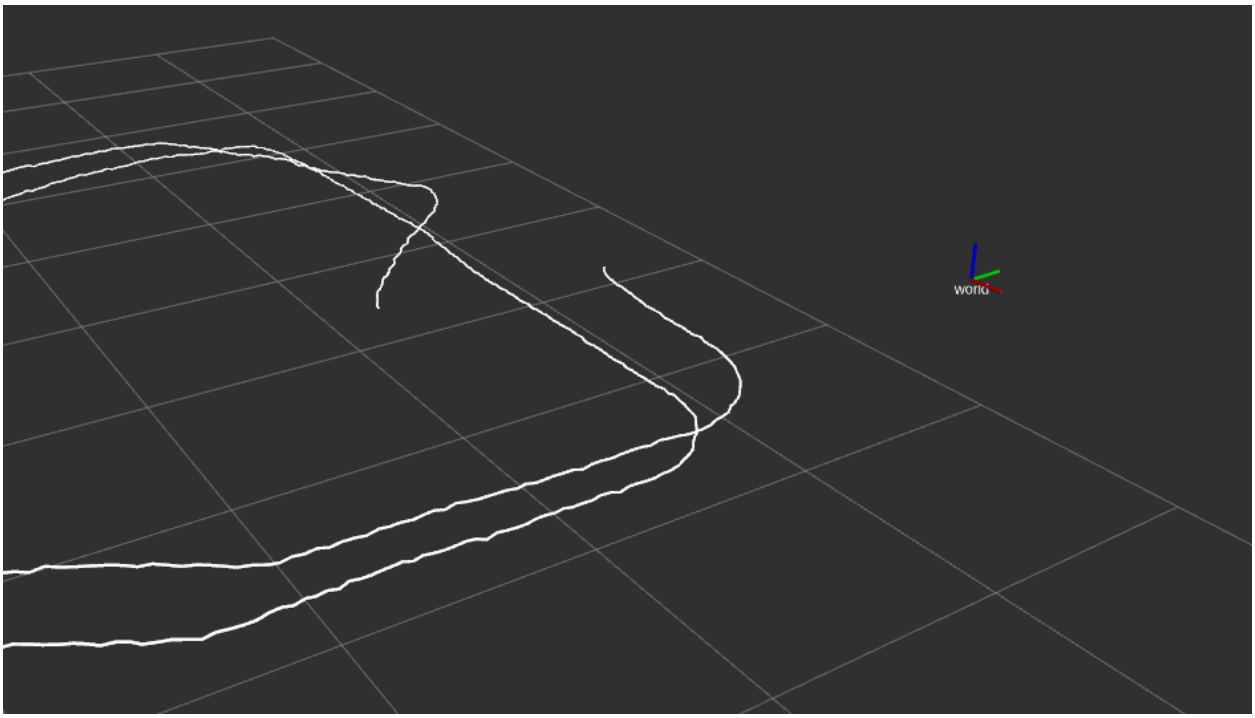
The entire transformation between moving frames can be visualized with the *ReferenceFrameMarkerPublisher* which publishes the translation of all valid timestamps as a `visualization_msgs/Marker` message on the `/<frame_name>/path` topic:

[15]:
```
marker_publisher = rbm.ros.ReferenceFrameMarkerPublisher("head", base="world")
marker_publisher.publish()
```

[16]:
```
import rospy

rospy.get_published_topics("/head")
```

[16]:
```
[['/head/path', 'visualization_msgs/Marker']]
```

RViz should now show the trajectory of the head frame as a white line:



Next, we will load data from the Pupil Core eye tracker to demonstrate how multiple moving frames can be visualized. With the *load_dataset* method, the data is imported as an `xarray.Dataset` and `cache=True` enables local caching in the netCDF4 format for faster loading.

[17]:
```
with reader:
    left_eye = reader.load_dataset("/pupil/left_eye/transform", cache=True)

left_eye = left_eye.dropna("time")
```

The transform between the T265 and the Pupil Core scene camera was determined in the study with a camera calibration routine and is provided here as hard-coded parameters:

[18]:
```
t_t265_pupil = (24.5e-3, -29e-3, 0.0)
r_t265_pupil = (-0.00125, -1.0, 6.3463e-05, 3.977e-06)

rbm.register_frame(
    "pupil/scene_cam",
    parent="t265/tracker",
    translation=t_t265_pupil,
    rotation=r_t265_pupil,
)
```

The reference frame of the left eye is constructed from the Dataset:

[19]:
```
rbm.ReferenceFrame.from_dataset(
    left_eye,
    "position",
    "orientation",
    "time",
    parent="pupil/scene_cam",
    name="pupil/left_eye",
```

(continues on next page)

(continued from previous page)

```
).register()
```

The eye tracker data uses yet another coordinate convention (x right, y down, z forward) which we need to take into account when visualizing the eye frame:

[20]:
```
R_PUPIL_ROS = np.array([[0.0, -1.0, 0.0], [0.0, 0.0, -1.0], [1.0, 0.0, 0.0]])

rbm.ReferenceFrame.from_rotation_matrix(
    R_PUPIL_ROS, parent="pupil/left_eye", name="left_eye"
).register(update=True)
```

[21]:
```
rbm.render_tree("world")
```

```
world
└── t265/world
    └── t265/tracker
        ├── head
        └── pupil/scene_cam
            └── pupil/left_eye
                └── left_eye
```

Now we create another broadcaster for the transform between head and eye frame. With `publish_pose=True`, the broadcaster also publishes a `geometry_msgs/PoseStamped` message on the `/<frame_name>/pose` topic. This message can be visualized in RViz as an arrow which avoids cluttering for frames that are close together. In order to publish messages in sync with the world/head transform, we specify `subscribe="head"`. This way, the broadcaster checks for new messages on the `/tf` topic where the `child_frame_id` is `"head"` and publishes its own transform with the closest timestamp:

[22]:
```
tf_head_left_eye = rbm.ros.ReferenceFrameTransformBroadcaster(
    "left_eye", base="head", publish_pose=True, subscribe="head"
)
```

Calling the *spin* method dispatches the broadcaster to a separate thread where it will keep checking for new world/head transforms:

[23]:
```
tf_head_left_eye.spin()
```

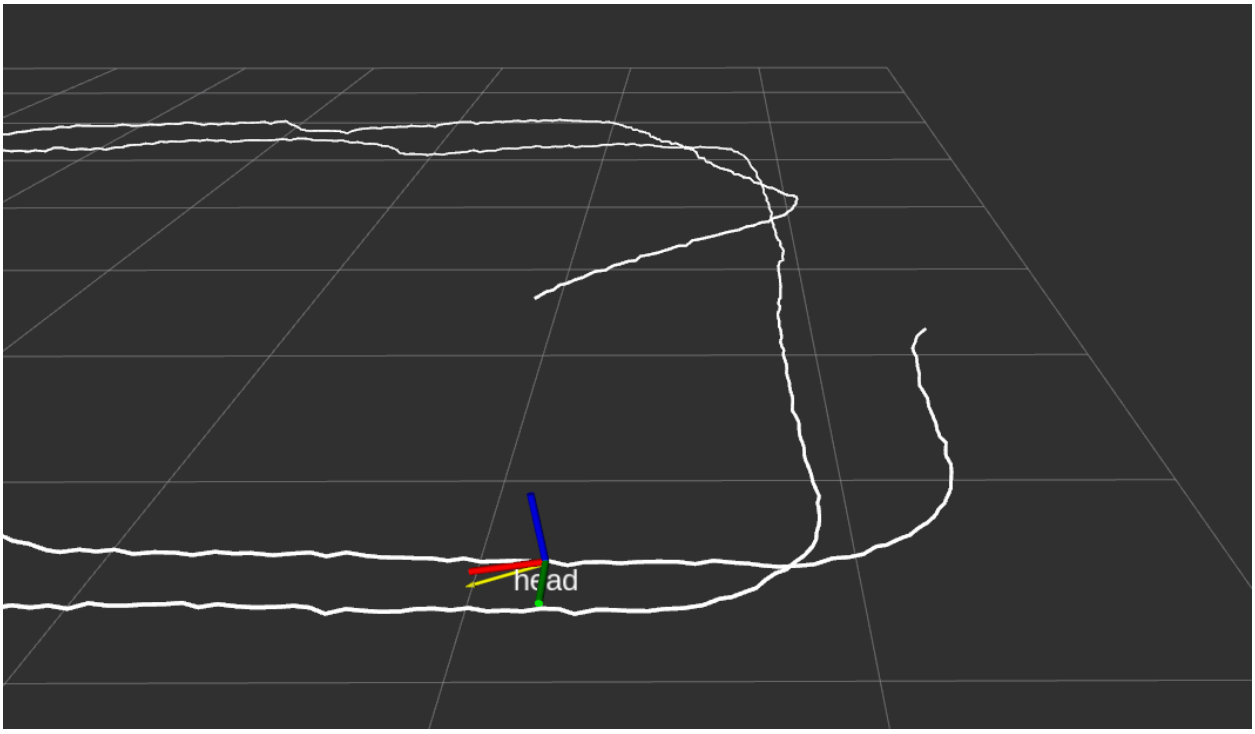
Finally, the *play_publisher* method provides a notebook widget to play back data from a broadcaster. With `step=2` it broadcasts every second transform and with `speed=0.5` the data is played back at half the recorded speed:

[24]:
```
rbm.ros.play_publisher(tf_world_head, step=2, speed=0.5)
```

```
HBox(children=(IntSlider(value=0, description='Index', max=66628), Button(description='',
↪ layout=Layout(widt...
```

```
Output()
```

After pressing the play button, you should see the head frame moving along the white path, with the eye-in-head pose drawn as a yellow arrow:



Finally, if you have started the ROS master from Python, you should shut it down at the end:

[25]:
```
master.shutdown()
```

```
[rosout-1] killing on exit
[master] killing on exit
```

1.7 API Reference

1.7.1 Top-level functions

Module: `rigid_body_motion`

Transformations

transform_vectors	Transform an array of vectors between reference frames.
transform_points	Transform an array of points between reference frames.
transform_quaternions	Transform an array of quaternions between reference frames.
transform_coordinates	Transform motion between coordinate systems.
transform_angular_velocity	Transform an array of angular velocities between frames.
transform_linear_velocity	Transform an array of linear velocities between frames.

rigid_body_motion.transform_vectors

rigid_body_motion.**transform_vectors**(*arr*, *into*, *outof=None*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *return_timestamps=False*)

Transform an array of vectors between reference frames.

Parameters

arr: array_like The array to transform.

into: str or ReferenceFrame ReferenceFrame instance or name of a registered reference frame in which the array will be represented after the transformation.

outof: str or ReferenceFrame, optional ReferenceFrame instance or name of a registered reference frame in which the array is currently represented. Can be omitted if the array is a DataArray whose `attrs` contain a "representation_frame" entry with the name of a registered frame.

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the vectors.

axis: int, optional The axis of the array representing the spatial coordinates of the vectors. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the vectors, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the vectors. Defaults to the first axis of the array.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like The timestamps after the transformation.

See also:

transform_quaternions, *transform_points*, *ReferenceFrame*

rigid_body_motion.transform_points

rigid_body_motion.**transform_points**(*arr*, *into*, *outof=None*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *return_timestamps=False*)

Transform an array of points between reference frames.

Parameters

arr: array_like The array to transform.

into: str or ReferenceFrame ReferenceFrame instance or name of a registered reference frame in which the array will be represented after the transformation.

outof: str or ReferenceFrame, optional ReferenceFrame instance or name of a registered reference frame which is the current reference frame of the array. Can be omitted if the array is

a DataArray whose `attrs` contain a "reference_frame" entry with the name of a registered frame.

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the points.

axis: int, optional The axis of the array representing the spatial coordinates of the points. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the points, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the points. Defaults to the first axis of the array.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like The timestamps after the transformation.

See also:

transform_vectors, *transform_quaternions*, *ReferenceFrame*

rigid_body_motion.transform_quaternions

rigid_body_motion.**transform_quaternions**(*arr*, *into*, *outof=None*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *return_timestamps=False*)

Transform an array of quaternions between reference frames.

Parameters

arr: array_like The array to transform.

into: str or ReferenceFrame ReferenceFrame instance or name of a registered reference frame in which the array will be represented after the transformation.

outof: str or ReferenceFrame, optional ReferenceFrame instance or name of a registered reference frame which is the current reference frame of the array. Can be omitted if the array is a DataArray whose `attrs` contain a "reference_frame" entry with the name of a registered frame.

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the quaternions.

axis: int, optional The axis of the array representing the spatial coordinates of the quaternions. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the quaternions, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the quaternions. Defaults to the first axis of the array.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like The timestamps after the transformation.

See also:

transform_vectors, *transform_points*, *ReferenceFrame*

rigid_body_motion.transform_coordinates

rigid_body_motion.**transform_coordinates**(*arr*, *into*, *outof=None*, *dim=None*, *axis=None*, *replace_dim=True*)

Transform motion between coordinate systems.

Parameters

arr: array_like The array to transform.

into: str The name of a coordinate system in which the array will be represented after the transformation.

outof: str, optional The name of a coordinate system in which the array is currently represented. Can be omitted if the array is a DataArray whose `attrs` contain a "coordinate_system" entry with the name of a valid coordinate system.

dim: str, optional If the array is a DataArray, the name of the dimension representing the coordinates of the motion.

axis: int, optional The axis of the array representing the coordinates of the motion. Defaults to the last axis of the array.

replace_dim: bool, default True If True and the array is a DataArray, replace the dimension representing the coordinates by a new dimension that describes the new coordinate system and its axes (e.g. `cartesian_axis: [x, y, z]`). All coordinates that contained the original dimension will be dropped.

Returns

arr_transformed: array_like The transformed array.

See also:

cartesian_to_polar, *polar_to_cartesian*, *cartesian_to_spherical*

spherical_to_cartesian

rigid_body_motion.transform_angular_velocity

rigid_body_motion.**transform_angular_velocity**(*arr*, *into*, *outof=None*, *what='reference_frame'*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *cutoff=None*, *return_timestamps=False*)

Transform an array of angular velocities between frames.

The array represents the velocity of a moving body or frame wrt a reference frame, expressed in a representation frame.

The transformation changes either the reference frame, the moving frame or the representation frame of the velocity from this frame to another. In either case, it is assumed that the array is represented in the frame that is being changed and will be represented in the new frame after the transformation.

When transforming the reference frame R to a new frame R' while keeping the moving frame M fixed, the transformed velocity is calculated according to the formula:

$$\omega_{M/R'} = \omega_{M/R} + \omega_{R/R'}$$

When transforming the moving frame M to a new frame M' while keeping the reference frame R fixed, the transformed velocity is calculated according to the formula:

$$\omega_{M'/R} = \omega_{M/R} + \omega_{M'/M}$$

Parameters

arr: array_like The array to transform.

into: str or ReferenceFrame The target reference frame.

outof: str or ReferenceFrame, optional The source reference frame. Can be omitted if the array is a DataArray whose `attrs` contain a "representation_frame", "reference_frame" or "moving_frame" entry with the name of a registered frame (depending on what you want to transform, see *what*).

what: str What frame of the velocity to transform. Can be "reference_frame", "moving_frame" or "representation_frame".

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the velocities.

axis: int, optional The axis of the array representing the spatial coordinates of the velocities. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the velocities, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the velocities. Defaults to the first axis of the array.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the twist estimation as a fraction of the Nyquist frequency.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like The timestamps after the transformation.

See also:

transform_linear_velocity, *transform_vectors*, *transform_quaternions*

transform_points, *ReferenceFrame*

rigid_body_motion.transform_linear_velocity

rigid_body_motion.**transform_linear_velocity**(*arr*, *into*, *outof=None*, *what='reference_frame'*, *moving_frame=None*, *reference_frame=None*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *cutoff=None*, *outlier_thresh=None*, *return_timestamps=False*)

Transform an array of linear velocities between frames.

The array represents the velocity of a moving body or frame wrt a reference frame, expressed in a representation frame.

The transformation changes either the reference frame, the moving frame or the representation frame of the velocity from this frame to another. In either case, it is assumed that the array is represented in the frame that is being changed and will be represented in the new frame after the transformation.

When transforming the reference frame R to a new frame R' while keeping the moving frame M fixed, the transformed velocity is calculated according to the formula:

$$v_{M/R'} = v_{M/R} + v_{R/R'} + \omega_{R/R'} \times t_{M/R}$$

When transforming the moving frame M to a new frame M' while keeping the reference frame R fixed, the transformed velocity is calculated according to the formula:

$$v_{M'/R} = v_{M/R} + v_{M'/M} + \omega_{M/R} \times t_{M'/M}$$

Parameters

arr: array_like The array to transform.

into: str or ReferenceFrame The target reference frame.

outof: str or ReferenceFrame, optional The source reference frame. Can be omitted if the array is a DataArray whose `attrs` contain a "representation_frame", "reference_frame" or "moving_frame" entry with the name of a registered frame (depending on what you want to transform, see *what*).

what: str What frame of the velocity to transform. Can be "reference_frame", "moving_frame" or "representation_frame".

moving_frame: str or ReferenceFrame, optional The moving frame when transforming the reference frame of the velocity.

reference_frame: str or ReferenceFrame, optional The reference frame when transforming the moving frame of the velocity.

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the velocities.

axis: int, optional The axis of the array representing the spatial coordinates of the velocities. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the velocities, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the velocities. Defaults to the first axis of the array.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the twist estimation as a fraction of the Nyquist frequency.

outlier_thresh: float, optional Some SLAM-based trackers introduce position corrections when a new camera frame becomes available. This introduces outliers in the linear velocity estimate. The estimation algorithm used here can suppress these outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values. For measurements from the Intel RealSense T265 tracker, set this value to 1e-3.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like The timestamps after the transformation.

See also:

transform_angular_velocity, *transform_vectors*, *transform_quaternions*

transform_points, *ReferenceFrame*

Reference Frames

ReferenceFrame	A three-dimensional reference frame.
register_frame	Register a new reference frame in the registry.
deregister_frame	Remove a reference frame from the registry.
clear_registry	Clear the reference frame registry.
render_tree	Render a reference frame tree.

rigid_body_motion.ReferenceFrame

class rigid_body_motion.**ReferenceFrame**(*name=None*, *parent=None*, *translation=None*, *rotation=None*, *timestamps=None*, *inverse=False*, *discrete=False*)

A three-dimensional reference frame.

__init__(*name=None*, *parent=None*, *translation=None*, *rotation=None*, *timestamps=None*, *inverse=False*, *discrete=False*)

Constructor.

Parameters

name: str, optional The name of this reference frame.

parent: str or ReferenceFrame, optional The parent reference frame. If str, the frame will be looked up in the registry under that name. If not specified, this frame will be a root node of a new reference frame tree.

translation: array_like, optional The translation of this frame wrt the parent frame. Not applicable if there is no parent frame.

rotation: array_like, optional The rotation of this frame wrt the parent frame. Not applicable if there is no parent frame.

timestamps: array_like, optional The timestamps for translation and rotation of this frame. Not applicable if this is a static reference frame.

inverse: bool, default False If True, invert the transform wrt the parent frame, i.e. the translation and rotation are specified for the parent frame wrt this frame.

discrete: bool, default False If True, transformations with timestamps are assumed to be events. Instead of interpolating between timestamps, transformations are fixed between their timestamp and the next one.

Methods

__init__([name, parent, translation, ...])	Constructor.
deregister()	Remove this frame from the registry.
from_dataset(ds, translation, rotation, ...)	Construct a reference frame from a Dataset.
from_rotation_dataarray(da, timestamps, parent)	Construct a reference frame from a rotation DataArray.
from_rotation_matrix(mat, parent[, name, ...])	Construct a static reference frame from a rotation matrix.
from_translation_dataarray(da, timestamps, ...)	Construct a reference frame from a translation DataArray.
get_transformation(to_frame)	Alias for lookup_transform.
iter_path_reverse()	Iterate up the tree from the current node.
lookup_angular_velocity([reference, ...])	Estimate angular velocity of this frame wrt a reference.
lookup_linear_velocity([reference, ...])	Estimate linear velocity of this frame wrt a reference.
lookup_transform(to_frame)	Look up the transformation from this frame to another.
lookup_twist([reference, represent_in, ...])	Estimate linear and angular velocity of this frame wrt a reference.
register([update])	Register this frame in the registry.
transform_angular_velocity(arr, to_frame[, ...])	Transform array of angular velocities from this frame to another.
transform_linear_velocity(arr, to_frame[, ...])	Transform array of linear velocities from this frame to another.
transform_points(arr, to_frame[, axis, ...])	Transform array of points from this frame to another.
transform_quaternions(arr, to_frame[, axis, ...])	Transform array of quaternions from this frame to another.
transform_vectors(arr, to_frame[, axis, ...])	Transform array of vectors from this frame to another.

Attributes

ancestors	All parent nodes and their parent nodes.
anchestors	All parent nodes and their parent nodes - see ancestors.
children	All child nodes.
depth	Number of edges to the root *Node*.
descendants	All child nodes and all their child nodes.
height	Number of edges on the longest path to a leaf *Node*.
is_leaf	*Node* has no children (External Node).
is_root	*Node* is tree root.
leaves	Tuple of all leaf nodes.
parent	Parent Node.
path	Path of this *Node*.
root	Tree Root Node.
separator	The NodeMixin class extends any Python class to a tree node.
siblings	Tuple of nodes with the same parent.

rigid_body_motion.register_frame

rigid_body_motion.**register_frame**(*name*, *parent=None*, *translation=None*, *rotation=None*, *timestamps=None*, *inverse=False*, *discrete=False*, *update=False*)

Register a new reference frame in the registry.

Parameters

name: str The name of the reference frame.

parent: str or ReferenceFrame, optional The parent reference frame. If str, the frame will be looked up in the registry under that name. If not specified, this frame will be a root node of a new reference frame tree.

translation: array_like, optional The translation of this frame wrt the parent frame. Not applicable if there is no parent frame.

rotation: array_like, optional The rotation of this frame wrt the parent frame. Not applicable if there is no parent frame.

timestamps: array_like, optional The timestamps for translation and rotation of this frame. Not applicable if this is a static reference frame.

inverse: bool, default False If True, invert the transform wrt the parent frame, i.e. the translation and rotation are specified for the parent frame wrt this frame.

discrete: bool, default False If True, transformations with timestamps are assumed to be events. Instead of interpolating between timestamps, transformations are fixed between their timestamp and the next one.

update: bool, default False If True, overwrite if there is a frame with the same name in the registry.

rigid_body_motion.deregister_frame

rigid_body_motion.**deregister_frame**(*name*)
Remove a reference frame from the registry.

Parameters

name: str The name of the reference frame.

rigid_body_motion.clear_registry

rigid_body_motion.**clear_registry**()
Clear the reference frame registry.

rigid_body_motion.render_tree

rigid_body_motion.**render_tree**(*root*)
Render a reference frame tree.

Parameters

root: str or ReferenceFrame The root of the rendered tree.

Coordinate Systems

cartesian_to_polar	Transform cartesian to polar coordinates in two dimensions.
polar_to_cartesian	Transform polar to cartesian coordinates in two dimensions.
cartesian_to_spherical	Transform cartesian to spherical coordinates in three dimensions.
spherical_to_cartesian	Transform spherical to cartesian coordinates in three dimensions.

rigid_body_motion.cartesian_to_polar

rigid_body_motion.**cartesian_to_polar**(*arr*, *axis=- 1*)
Transform cartesian to polar coordinates in two dimensions.

Parameters

arr [array_like] Input array.

axis [int, default -1] Axis of input array representing x and y in cartesian coordinates. Must be of length 2.

Returns

arr_polar [array_like] Output array.

rigid_body_motion.polar_to_cartesian

rigid_body_motion.**polar_to_cartesian**(*arr*, *axis=- 1*)

Transform polar to cartesian coordinates in two dimensions.

Parameters

arr [array_like] Input array.

axis [int, default -1] Axis of input array representing r and phi in polar coordinates. Must be of length 2.

Returns

arr_cartesian [array_like] Output array.

rigid_body_motion.cartesian_to_spherical

rigid_body_motion.**cartesian_to_spherical**(*arr*, *axis=- 1*)

Transform cartesian to spherical coordinates in three dimensions.

The spherical coordinate system is defined according to ISO 80000-2.

Parameters

arr [array_like] Input array.

axis [int, default -1] Axis of input array representing x, y and z in cartesian coordinates. Must be of length 3.

Returns

arr_spherical [array_like] Output array.

rigid_body_motion.spherical_to_cartesian

rigid_body_motion.**spherical_to_cartesian**(*arr*, *axis=- 1*)

Transform spherical to cartesian coordinates in three dimensions.

The spherical coordinate system is defined according to ISO 80000-2.

Parameters

arr [array_like] Input array.

axis [int, default -1] Axis of input array representing r, theta and phi in spherical coordinates. Must be of length 3.

Returns

arr_cartesian [array_like] Output array.

Lookups

lookup_transform	Look up transformation from one frame to another.
lookup_pose	Look up pose of one frame wrt a reference.
lookup_twist	Estimate linear and angular velocity of a frame wrt a reference.
lookup_linear_velocity	Estimate linear velocity of a frame wrt a reference.
lookup_angular_velocity	Estimate angular velocity of a frame wrt a reference.

rigid_body_motion.lookup_transform

rigid_body_motion.**lookup_transform**(*outof*, *into*, *as_dataset=False*, *return_timestamps=False*)

Look up transformation from one frame to another.

The transformation is a rotation r followed by a translation t which, when applied to a point expressed wrt the base frame B, yields that point wrt the target frame T:

$$p_T = rot(r, p_B) + t$$

Parameters

outof: str or ReferenceFrame Base frame of the transformation.

into: str or ReferenceFrame Target frame of the transformation.

as_dataset: bool, default False If True, return an xarray.Dataset. Otherwise, return a tuple of translation and rotation.

return_timestamps: bool, default False If True, and *as_dataset* is False, also return the timestamps of the lookup.

Returns

translation, rotation: each numpy.ndarray Translation and rotation of transformation between the frames, if *as_dataset* is False.

timestamps: numpy.ndarray Corresponding timestamps of the lookup if *return_timestamps* is True.

ds: xarray.Dataset The above arrays as an xarray.Dataset, if *as_dataset* is True.

rigid_body_motion.lookup_pose

rigid_body_motion.**lookup_pose**(*frame*, *reference*, *as_dataset=False*, *return_timestamps=False*)

Look up pose of one frame wrt a reference.

Parameters

frame: str or ReferenceFrame Frame for which to look up the pose.

reference: str or ReferenceFrame Reference frame of the pose.

as_dataset: bool, default False If True, return an xarray.Dataset. Otherwise, return a tuple of position and orientation.

return_timestamps: bool, default False If True, and *as_dataset* is False, also return the timestamps of the lookup.

Returns

position, orientation: each numpy.ndarray Position and orientation of the pose between the frames, if *as_dataset* is False.

timestamps: numpy.ndarray Corresponding timestamps of the lookup if *return_timestamps* is True.

ds: xarray.Dataset The above arrays as an xarray.Dataset, if *as_dataset* is True.

rigid_body_motion.lookup_twist

rigid_body_motion.**lookup_twist**(*frame*, *reference=None*, *represent_in=None*, *outlier_thresh=None*, *cutoff=None*, *mode='quaternion'*, *as_dataset=False*, *return_timestamps=False*)

Estimate linear and angular velocity of a frame wrt a reference.

Parameters

frame: str or ReferenceFrame The reference frame whose twist is estimated.

reference: str or ReferenceFrame, optional The reference frame wrt which the twist is estimated. Defaults to the parent frame of the moving frame.

represent_in: str or ReferenceFrame, optional The reference frame in which the twist is represented. Defaults to the reference frame.

outlier_thresh: float, optional Some SLAM-based trackers introduce position corrections when a new camera frame becomes available. This introduces outliers in the linear velocity estimate. The estimation algorithm used here can suppress these outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values. For measurements from the Intel RealSense T265 tracker, set this value to 1e-3.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the estimation as a fraction of the Nyquist frequency.

mode: str, default "quaternion" If "quaternion", compute the angular velocity from the quaternion derivative. If "rotation_vector", compute the angular velocity from the gradient of the axis-angle representation of the rotations.

as_dataset: bool, default False If True, return an xarray.Dataset. Otherwise, return a tuple of linear and angular velocity.

return_timestamps: bool, default False If True, and *as_dataset* is False, also return the timestamps of the lookup.

Returns

linear, angular: each numpy.ndarray Linear and angular velocity of moving frame wrt reference frame, represented in representation frame, if *as_dataset* is False.

timestamps: numpy.ndarray Corresponding timestamps of the lookup if *return_timestamps* is True.

ds: xarray.Dataset The above arrays as an xarray.Dataset, if *as_dataset* is True.

rigid_body_motion.lookup_linear_velocity

rigid_body_motion.**lookup_linear_velocity**(*frame*, *reference=None*, *represent_in=None*, *outlier_thresh=None*, *cutoff=None*, *as_dataarray=False*, *return_timestamps=False*)

Estimate linear velocity of a frame wrt a reference.

Parameters

frame: str or ReferenceFrame The reference frame whose velocity is estimated.

reference: str or ReferenceFrame, optional The reference frame wrt which the velocity is estimated. Defaults to the parent frame of the moving frame.

represent_in: str or ReferenceFrame, optional The reference frame in which the twist is represented. Defaults to the reference frame.

outlier_thresh: float, optional Some SLAM-based trackers introduce position corrections when a new camera frame becomes available. This introduces outliers in the linear velocity estimate. The estimation algorithm used here can suppress these outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values. For measurements from the Intel RealSense T265 tracker, set this value to 1e-3.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the estimation as a fraction of the Nyquist frequency.

as_dataarray: bool, default False If True, return an xarray.DataArray.

return_timestamps: bool, default False If True and *as_dataarray* is False, also return the timestamps of the lookup.

Returns

linear: numpy.ndarray or xarray.DataArray Linear velocity of moving frame wrt reference frame, represented in representation frame.

timestamps: numpy.ndarray Corresponding timestamps of the lookup if *return_timestamps* is True.

rigid_body_motion.lookup_angular_velocity

rigid_body_motion.**lookup_angular_velocity**(*frame*, *reference=None*, *represent_in=None*, *outlier_thresh=None*, *cutoff=None*, *mode='quaternion'*, *as_dataarray=False*, *return_timestamps=False*)

Estimate angular velocity of a frame wrt a reference.

Parameters

frame: str or ReferenceFrame The reference frame whose velocity is estimated.

reference: str or ReferenceFrame, optional The reference frame wrt which the velocity is estimated. Defaults to the parent frame of the moving frame.

represent_in: str or ReferenceFrame, optional The reference frame in which the twist is represented. Defaults to the reference frame.

outlier_thresh: float, optional Suppress samples where the norm of the second-order differences of the rotation is above *outlier_thresh* and interpolate the missing values.

cutoff: float, optional Frequency of a low-pass filter applied to angular and angular velocity after the estimation as a fraction of the Nyquist frequency.

mode: str, default "quaternion" If "quaternion", compute the angular velocity from the quaternion derivative. If "rotation_vector", compute the angular velocity from the gradient of the axis-angle representation of the rotations.

as_dataarray: bool, default False If True, return an xarray.DataArray.

return_timestamps: bool, default False If True and *as_dataarray* is False, also return the timestamps of the lookup.

Returns

angular: numpy.ndarray or xarray.DataArray Angular velocity of moving frame wrt reference frame, represented in representation frame.

timestamps: numpy.ndarray Corresponding timestamps of the lookup if *return_timestamps* is True.

Estimators

estimate_linear_velocity	Estimate linear velocity from a time series of translation.
estimate_angular_velocity	Estimate angular velocity from a time series of rotations.
shortest_arc_rotation	Estimate the shortest-arc rotation between two arrays of vectors.
best_fit_rotation	Least-squares best-fit rotation between two arrays of vectors.
best_fit_transform	Least-squares best-fit transform between two arrays of vectors.
iterative_closest_point	Iterative closest point algorithm matching two arrays of vectors.

rigid_body_motion.estimate_linear_velocity

rigid_body_motion.**estimate_linear_velocity**(*arr*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *outlier_thresh=None*, *cutoff=None*)

Estimate linear velocity from a time series of translation.

Parameters

arr: array_like Array of translations.

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the points.

axis: int, optional The axis of the array representing the spatial coordinates of the points. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the points, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the points. Defaults to the first axis of the array.

cutoff: float, optional Frequency of a low-pass filter applied to the linear velocity after the estimation as a fraction of the Nyquist frequency.

outlier_thresh: float, optional Some SLAM-based trackers introduce position corrections when a new camera frame becomes available. This introduces outliers in the linear velocity estimate. The estimation algorithm used here can suppress these outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values. For measurements from the Intel RealSense T265 tracker, set this value to 1e-3.

Returns

linear: array_like Array of linear velocities.

rigid_body_motion.estimate_angular_velocity

rigid_body_motion.**estimate_angular_velocity**(*arr*, *dim=None*, *axis=None*, *timestamps=None*, *time_axis=None*, *mode='quaternion'*, *outlier_thresh=None*, *cutoff=None*)

Estimate angular velocity from a time series of rotations.

Parameters

arr: array_like Array of rotations, expressed in quaternions.

dim: str, optional If the array is a DataArray, the name of the dimension representing the spatial coordinates of the quaternions.

axis: int, optional The axis of the array representing the spatial coordinates of the quaternions. Defaults to the last axis of the array.

timestamps: array_like or str, optional The timestamps of the quaternions, corresponding to the *time_axis* of the array. If str and the array is a DataArray, the name of the coordinate with the timestamps. The axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

time_axis: int, optional The axis of the array representing the timestamps of the quaternions. Defaults to the first axis of the array.

mode: str, default "quaternion" If "quaternion", compute the angular velocity from the quaternion derivative. If "rotation_vector", compute the angular velocity from the gradient of the axis-angle representation of the rotations.

outlier_thresh: float, optional Suppress samples where the norm of the second-order differences of the rotation is above *outlier_thresh* and interpolate the missing values.

cutoff: float, optional Frequency of a low-pass filter applied to the angular velocity after the estimation as a fraction of the Nyquist frequency.

Returns

angular: array_like Array of angular velocities.

rigid_body_motion.shortest_arc_rotation

rigid_body_motion.**shortest_arc_rotation**(*v1*, *v2*, *dim=None*, *axis=None*)

Estimate the shortest-arc rotation between two arrays of vectors.

This method computes the rotation *r* that satisfies:

$$v_2/||v_2|| = rot(r, v_1)/||v_1||$$

Parameters

v1: array_like, shape (…, 3, …) The first array of vectors.

v2: array_like, shape (…, 3, …) The second array of vectors.

dim: str, optional If the first array is a DataArray, the name of the dimension representing the spatial coordinates of the vectors.

axis: int, optional The axis of the arrays representing the spatial coordinates of the vectors. Defaults to the last axis of the arrays.

Returns

rotation: array_like, shape (…, 4, …) The quaternion representation of the shortest-arc rotation.

rigid_body_motion.best_fit_rotation

rigid_body_motion.**best_fit_rotation**(*v1*, *v2*, *dim=None*, *axis=None*)

Least-squares best-fit rotation between two arrays of vectors.

Finds the rotation *r* that minimizes:

$$||v_2 - rot(r, v_1)||$$

Parameters

v1: array_like, shape (…, 3, …) The first array of vectors.

v2: array_like, shape (…, 3, …) The second array of vectors.

dim: str, optional If the first array is a DataArray, the name of the dimension representing the spatial coordinates of the vectors.

axis: int, optional The axis of the arrays representing the spatial coordinates of the vectors. Defaults to the last axis of the arrays.

Returns

rotation: array_like, shape (4,) Rotation of transform.

See also:

iterative_closest_point, *best_fit_transform*

References

Adapted from https://github.com/ClayFlannigan/icp

rigid_body_motion.best_fit_transform

rigid_body_motion.**best_fit_transform**(*v1*, *v2*, *dim=None*, *axis=None*)

Least-squares best-fit transform between two arrays of vectors.

Finds the rotation *r* and the translation *t* that minimize:

$$||v_2 - (rot(r, v_1) + t)||$$

Parameters

v1: array_like, shape (..., 3, ...) The first array of vectors.

v2: array_like, shape (..., 3, ...) The second array of vectors.

dim: str, optional If the first array is a DataArray, the name of the dimension representing the spatial coordinates of the vectors.

axis: int, optional The axis of the arrays representing the spatial coordinates of the vectors. Defaults to the last axis of the arrays.

Returns

translation: array_like, shape (3,) Translation of transform.

rotation: array_like, shape (4,) Rotation of transform.

See also:

iterative_closest_point, *best_fit_rotation*

References

Adapted from https://github.com/ClayFlannigan/icp

rigid_body_motion.iterative_closest_point

rigid_body_motion.**iterative_closest_point**(*v1*, *v2*, *dim=None*, *axis=None*, *init_transform=None*, *max_iterations=20*, *tolerance=0.001*)

Iterative closest point algorithm matching two arrays of vectors.

Finds the rotation *r* and the translation *t* such that:

$$v_2 \simeq rot(r, v_1) + t$$

Parameters

v1: array_like, shape (..., 3, ...) The first array of vectors.

v2: array_like, shape (..., 3, ...) The second array of vectors.

dim: str, optional If the first array is a DataArray, the name of the dimension representing the spatial coordinates of the vectors.

axis: int, optional The axis of the arrays representing the spatial coordinates of the vectors. Defaults to the last axis of the arrays.

init_transform: tuple, optional Initial guess as (translation, rotation) tuple.

max_iterations: int, default 20 Maximum number of iterations.

tolerance: float, default 1e-3 Abort if the mean distance error between the transformed arrays does not improve by more than this threshold between iterations.

Returns

translation: array_like, shape (3,) Translation of transform.

rotation: array_like, shape (4,) Rotation of transform.

See also:

best_fit_transform, *best_fit_rotation*

Notes

For points with known correspondences (e.g. timeseries of positions), it is recommended to interpolate the points to a common sampling base and use the *best_fit_transform* method.

References

Adapted from https://github.com/ClayFlannigan/icp

Utility functions

from_euler_angles	Construct quaternions from Euler angles.
qinv	Quaternion inverse.
qmul	Quaternion multiplication.
qmean	Quaternion mean.
qinterp	Quaternion interpolation.
rotate_vectors	Rotate an array of vectors by an array of quaternions.

rigid_body_motion.from_euler_angles

rigid_body_motion.**from_euler_angles**(*rpy=None*, *roll=None*, *pitch=None*, *yaw=None*, *axis=- 1*, *order='rpy'*, *return_quaternion=False*)

Construct quaternions from Euler angles.

This method differs from the method found in the quaternion package in that it is actually useful for converting commonly found Euler angle representations to quaternions.

Parameters

rpy: array-like, shape (..., 3, ...), optional Array with roll, pitch and yaw values. Mutually exclusive with *roll*, *pitch* and *yaw* arguments.

roll: array-like, optional Array with roll values. Mutually exclusive with *rpy* argument.

pitch: array-like, optional Array with pitch values. Mutually exclusive with *rpy* argument.

yaw: array-like, optional Array with yaw values. Mutually exclusive with *rpy* argument.

axis: int, default -1 Array axis representing RPY values of *rpy* argument.

order: str, default "rpy" Order of consecutively applied rotations. Defaults to roll -> pitch -> yaw.

return_quaternion: bool, default False If True, return result as quaternion dtype.

Returns

q: array-like Array with quaternions

rigid_body_motion.qinv

rigid_body_motion.**qinv**(*q*, *qaxis=- 1*)

Quaternion inverse.

Parameters

q: array_like Array containing quaternions whose inverse is to be computed. Its dtype can be quaternion, otherwise *qaxis* specifies the axis representing the quaternions.

qaxis: int, default -1 If *q* is not quaternion dtype, axis of the quaternion array representing the coordinates of the quaternions.

Returns

qi: ndarray A new array containing the inverse values.

rigid_body_motion.qmul

rigid_body_motion.**qmul**(**q*, *qaxis=- 1*)

Quaternion multiplication.

Parameters

q: iterable of array_like Arrays containing quaternions to multiply. Their dtype can be quaternion, otherwise *qaxis* specifies the axis representing the quaternions.

qaxis: int, default -1 If *q* are not quaternion dtype, axis of the quaternion arrays representing the coordinates of the quaternions.

Returns

qm: ndarray A new array containing the multiplied quaternions.

rigid_body_motion.qmean

rigid_body_motion.**qmean**(*q*, *axis=None*, *qaxis=- 1*)

Quaternion mean.

Adapted from https://github.com/christophhagen/averaging-quaternions.

Parameters

q: array_like Array containing quaternions whose mean is to be computed. Its dtype can be quaternion, otherwise *qaxis* specifies the axis representing the quaternions.

axis: None or int or tuple of ints, optional Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

qaxis: int, default -1 If *q* is not quaternion dtype, axis of the quaternion array representing the coordinates of the quaternions.

Returns

qm: ndarray A new array containing the mean values.

rigid_body_motion.qinterp

rigid_body_motion.**qinterp**(*q*, *t_in*, *t_out*, *axis=0*, *qaxis=- 1*)

Quaternion interpolation.

Parameters

q: array_like Array containing quaternions to interpolate. Its dtype can be quaternion, otherwise *qaxis* specifies the axis representing the quaternions.

t_in: array_like Array of current sampling points of *q*.

t_out: array_like Array of desired sampling points of *q*.

axis: int, default 0 Axis along which the quaternions are interpolated.

qaxis: int, default -1 If *q* is not quaternion dtype, axis of the quaternion array representing the coordinates of the quaternions.

Returns

qi: ndarray A new array containing the interpolated values.

rigid_body_motion.rotate_vectors

rigid_body_motion.**rotate_vectors**(*q*, *v*, *axis=- 1*, *qaxis=- 1*, *one_to_one=True*)

Rotate an array of vectors by an array of quaternions.

Parameters

q: array_like Array of quaternions. Its dtype can be quaternion, otherwise *q_axis* specifies the axis representing the quaternions.

v: array_like The array of vectors to be rotated.

axis: int, default -1 The axis of the *v* array representing the coordinates of the vectors. Must have length 3.

qaxis: int, default -1 If *q* is not quaternion dtype, axis of the quaternion array representing the coordinates of the quaternions.

one_to_one: bool, default True If True, rotate each vector by a single quaternion. In this case, non-singleton dimensions of *q* and *v* must match. Otherwise, perform rotations for all combinations of *q* and *v*.

Returns

vr: array_like The array of rotated vectors. If *one_to_one=True* this array has the shape of all non-singleton dimensions in *q* and *v*. Otherwise, this array has shape *q.shape + v.shape*.

1.7.2 I/O functions

Module: `rigid_body_motion.io`

load_optitrack	Load rigid body poses from OptiTrack csv export.

rigid_body_motion.io.load_optitrack

`rigid_body_motion.io.`**`load_optitrack`**(*filepath*, *import_format='numpy'*)

Load rigid body poses from OptiTrack csv export.

Parameters

filepath: str or path-like Path to csv file.

import_format: {"numpy", "pandas", "xarray"}, default "numpy" Import format for rigid body poses. "numpy" returns a (position, orientation, timestamps) tuple for each body, "pandas" returns a DataFrame and "xarray" returns a Dataset.

Returns

data_dict: dict Dictionary with one entry for each rigid body. See `import_format` for the format of each entry.

1.7.3 Plotting

Module: `rigid_body_motion.plot`

reference_frame	Plot a 3D coordinate system representing a static reference frame.
points	Plot an array of 3D points.
quaternions	Plot an array of quaternions.
vectors	Plot an array of 3D vectors.

rigid_body_motion.plot.reference_frame

`rigid_body_motion.plot.`**`reference_frame`**(*frame*, *world_frame=None*, *ax=None*, *figsize=(6, 6)*, *arrow_len=1.0*)

Plot a 3D coordinate system representing a static reference frame.

Parameters

frame: str or ReferenceFrame The reference frame to plot. If str, the frame will be looked up in the registry under that name.

world_frame: str or ReferenceFrame, optional If specified, the world reference frame that defines the origin of the plot. If str, the frame will be looked up in the registry under that name.

ax: matplotlib.axes.Axes instance, optional If provided, plot the points onto these axes.

figsize: If *ax* is not provided, create a figure of this size.

arrow_len: Length of the arrows of the coordinate system.

Returns

arrows: list of Arrow3D A list of three arrows representing the plotted coordinate system.

rigid_body_motion.plot.points

rigid_body_motion.plot.**points**(*arr*, *ax=None*, *figsize=(6, 6)*, *fmt=''*, ***kwargs*)

Plot an array of 3D points.

Parameters

arr: array_like, shape (3,) or (N, 3) Array of 3D points to plot.

ax: matplotlib.axes.Axes instance, optional If provided, plot the points onto these axes.

figsize: If *ax* is not provided, create a figure of this size.

fmt: str, optional A format string, e.g. 'ro' for red circles.

kwargs: Additional keyword arguments passed to ax.plot().

Returns

lines: list of Line3D A list of lines representing the plotted data.

rigid_body_motion.plot.quaternions

rigid_body_motion.plot.**quaternions**(*arr*, *base=None*, *ax=None*, *figsize=(6, 6)*, ***kwargs*)

Plot an array of quaternions.

Parameters

arr: array_like, shape (4,) or (N, 4) Array of quaternions to plot.

base: array_like, shape (4,) or (N, 4), optional If provided, base points of the quaternions.

ax: matplotlib.axes.Axes instance, optional If provided, plot the points onto these axes.

figsize: If *ax* is not provided, create a figure of this size.

kwargs: Additional keyword arguments passed to ax.quiver().

Returns

lines: list of Line3DCollection A list of lines representing the plotted data.

rigid_body_motion.plot.vectors

rigid_body_motion.plot.**vectors**(*arr*, *base=None*, *ax=None*, *figsize=(6, 6)*, ***kwargs*)

Plot an array of 3D vectors.

Parameters

arr: array_like, shape (3,) or (N, 3) Array of 3D points to plot.

base: array_like, shape (3,) or (N, 3), optional If provided, base points of the vectors.

ax: matplotlib.axes.Axes instance, optional If provided, plot the points onto these axes.

figsize: If *ax* is not provided, create a figure of this size.

kwargs: Additional keyword arguments passed to ax.quiver().

Returns

lines: Line3DCollection A collection of lines representing the plotted data.

1.7.4 ROS interface

Module: `rigid_body_motion.ros`

RosbagReader	Reader for motion topics from rosbag files.
RosbagWriter	Writer for motion topics to rosbag files.
ReferenceFrameTransformBroadcaster	TF broadcaster for the transform of a reference frame wrt another.
ReferenceFrameMarkerPublisher	Publisher for the translation of a reference frame wrt another.
Transformer	Wrapper class for tf2_ros.Buffer.

rigid_body_motion.ros.RosbagReader

class rigid_body_motion.ros.**RosbagReader**(*bag_file*)

Bases: `object`

Reader for motion topics from rosbag files.

__init__(*bag_file*)

Constructor.

Parameters

bag_file: str Path to rosbag file.

Methods

__init__(bag_file)	Constructor.
export(topic[, output_file])	Export messages from topic as netCDF4 file.
get_topics_and_types()	Get topics and corresponding message types included in rosbag.
load_dataset(topic[, cache])	Load messages from topic as xarray.Dataset.
load_messages(topic)	Load messages from topic as dict.

export(*topic*, *output_file=None*)

Export messages from topic as netCDF4 file.

Parameters

topic: str Topic to read.

output_file: str, optional Path to output file. By default, the path to the bag file, but with a different extension depending on the export format.

get_topics_and_types()

Get topics and corresponding message types included in rosbag.

Returns

topics: dict Names of topics and corresponding message types included in the rosbag.

load_dataset(*topic*, *cache=False*)

Load messages from topic as xarray.Dataset.

Only nav_msgs/Odometry and geometry_msgs/TransformStamped topics are supported so far.

Parameters

topic: str Name of the topic to load.

cache: bool, default False If True, cache the dataset in `cache/<topic>.nc` in the same folder as the rosbag.

Returns

ds: xarray.Dataset Messages as dataset.

`load_messages`(*topic*)
Load messages from topic as dict.

Only nav_msgs/Odometry and geometry_msgs/TransformStamped topics are supported so far.

Parameters

topic: str Name of the topic to load.

Returns

messages: dict Dict containing arrays of timestamps and other message contents.

rigid_body_motion.ros.RosbagWriter

`class rigid_body_motion.ros.RosbagWriter`(*bag_file*)
Bases: `object`

Writer for motion topics to rosbag files.

`__init__`(*bag_file*)
Constructor.

Parameters

bag_file: str Path to rosbag file.

Methods

__init__(bag_file)	Constructor.
write_transform_stamped(timestamps, ...)	Write multiple geometry_msgs/TransformStamped messages.
write_transform_stamped_dataset(ds, topic, ...)	Write a dataset as geometry_msgs/TransformStamped messages.

`write_transform_stamped`(*timestamps*, *translation*, *rotation*, *topic*, *frame*, *child_frame*)
Write multiple geometry_msgs/TransformStamped messages.

Parameters

timestamps: array_like, shape (n_timestamps,) Array of timestamps.

translation: array_like, shape (n_timestamps, 3) Array of translations.

rotation: array_like, shape (n_timestamps, 4) Array of rotations.

topic: str Topic of the messages.

frame: str Parent frame of the transform.

child_frame: str Child frame of the transform.

write_transform_stamped_dataset(*ds*, *topic*, *frame*, *child_frame*, *timestamps='time'*, *translation='position'*, *rotation='orientation'*)

Write a dataset as geometry_msgs/TransformStamped messages.

Parameters

ds: xarray.Dataset Dataset containing timestamps, translation and rotation

topic: str Topic of the messages.

frame: str Parent frame of the transform.

child_frame: str Child frame of the transform.

timestamps: str, default 'time' Name of the dimension containing the timestamps.

translation: str, default 'position' Name of the variable containing the translation.

rotation: str, default 'orientation' Name of the variable containing the rotation.

rigid_body_motion.ros.ReferenceFrameTransformBroadcaster

class rigid_body_motion.ros.ReferenceFrameTransformBroadcaster(*frame*, *base=None*, *publish_pose=False*, *publish_twist=False*, *subscribe=False*, *twist_cutoff=None*, *twist_outlier_thresh=None*)

Bases: object

TF broadcaster for the transform of a reference frame wrt another.

__init__(*frame*, *base=None*, *publish_pose=False*, *publish_twist=False*, *subscribe=False*, *twist_cutoff=None*, *twist_outlier_thresh=None*)

Constructor.

Parameters

frame [str or ReferenceFrame] Reference frame for which to publish the transform.

base [str or ReferenceFrame, optional] Base reference wrt to which the transform is published. Defaults to the parent reference frame.

publish_pose [bool, default False] If True, also publish a PoseStamped message on the topic "/<frame>/pose".

publish_twist [bool, default False] If True, also publish a TwistStamped message with the linear and angular velocity of the frame wrt the base on the topic "/<frame>/twist".

subscribe [bool or str, default False] If True, subscribe to the "/tf" topic and publish transforms when messages are broadcast whose *child_frame_id* is the name of the base frame. If the frame is a moving reference frame, the transform whose timestamp is the closest to the broadcast timestamp is published. *subscribe* can also be a string, in which case this broadcaster will be listening for TF messages with this *child_frame_id*.

Methods

__init__(frame[, base, publish_pose, ...])	Constructor.
handle_incoming_msg(msg)	Publish on incoming message.
publish([idx])	Publish a transform message.
spin([block])	Continuously publish messages.
stop()	Stop publishing.

handle_incoming_msg(*msg*)
Publish on incoming message.

publish(*idx=None*)
Publish a transform message.

Parameters

idx [int, optional] Index of the transform to publish for a moving reference frame. Uses `self.idx` as default.

spin(*block=False*)
Continuously publish messages.

Parameters

block: bool, default False If True, this method will block until the publisher is stopped, e.g. by calling stop(). Otherwise, the main loop is dispatched to a separate thread which is returned by this function.

Returns

thread: threading.Thread If *block=True*, the Thread instance that runs the loop.

stop()
Stop publishing.

rigid_body_motion.ros.ReferenceFrameMarkerPublisher

class rigid_body_motion.ros.**ReferenceFrameMarkerPublisher**(*frame*, *base=None*, *topic=None*, *max_points=1000*, *publish_interval=0.0*, *scale=0.1*, *color='#ffffffff'*, *verbose=False*)

Bases: `rigid_body_motion.ros.visualization.BaseMarkerPublisher`

Publisher for the translation of a reference frame wrt another.

__init__(*frame*, *base=None*, *topic=None*, *max_points=1000*, *publish_interval=0.0*, *scale=0.1*, *color='#ffffffff'*, *verbose=False*)
Constructor.

Parameters

frame [str or ReferenceFrame] Reference frame for which to publish the translation.

base [str or ReferenceFrame, optional] Base reference wrt to which the translation is published. Defaults to the parent reference frame.

topic [str, optional] Name of the topic on which to publish. Defaults to "/<frame>/path".

max_points [int, default 1000] Maximum number of points to add to the marker. Actual translation array will be sub-sampled to this number of points.

publish_interval [float, default 0.0] Time in seconds between publishing when calling `spin`.

Methods

__init__(frame[, base, topic, max_points, ...])	Constructor.
get_ros3d_widget([ros, tf_client])	Get a ros3d.Marker widget to display in a ros3d.Viewer.
publish()	Publish a marker message.
spin([block])	Continuously publish messages.
stop()	Stop publishing.

get_ros3d_widget(*ros=None*, *tf_client=None*)
Get a ros3d.Marker widget to display in a ros3d.Viewer.

Parameters

ros [jupyros.ros3d.ROSConnection, optional] ros3d ROS connection instance.

tf_client [jupyros.ros3d.TFClient, optional] ros3d TF client instance.

Returns

jupyros.ros3d.Marker ros3d marker widget.

publish()
Publish a marker message.

spin(*block=False*)
Continuously publish messages.

Parameters

block: bool, default False If True, this method will block until the publisher is stopped, e.g. by calling stop(). Otherwise, the main loop is dispatched to a separate thread which is returned by this function.

Returns

thread: threading.Thread If *block=True*, the Thread instance that runs the loop.

stop()
Stop publishing.

rigid_body_motion.ros.Transformer

class rigid_body_motion.ros.**Transformer**(*cache_time=None*)
Bases: `object`

Wrapper class for tf2_ros.Buffer.

Can be constructed from a ReferenceFrame instance.

__init__(*cache_time=None*)
Constructor.

Parameters

cache_time [float, optional] Cache time of the buffer in seconds.

Methods

__init__([cache_time])	Constructor.
can_transform(target_frame, source_frame[, time])	Check if transform from source to target frame is possible.
from_reference_frame(reference_frame)	Construct Transformer instance from static reference frame tree.
lookup_transform(target_frame, source_frame)	Get the transform from the source frame to the target frame.
set_transform_static(reference_frame)	Add static transform from reference frame to buffer.
set_transforms(reference_frame)	Add transforms from moving reference frame to buffer.
transform_point(p, target_frame, source_frame)	Transform a point from the source frame to the target frame.
transform_pose(p, o, target_frame, source_frame)	Transform a pose from the source frame to the target frame.
transform_quaternion(q, target_frame, ...[, ...])	Transform a quaternion from the source frame to the target frame.
transform_vector(v, target_frame, source_frame)	Transform a vector from the source frame to the target frame.

can_transform(*target_frame*, *source_frame*, *time=0.0*)
Check if transform from source to target frame is possible.

Parameters

target_frame [str] Name of the frame to transform into.

source_frame [str] Name of the input frame.

time [float, default 0.0] Time at which to get the transform. (0 will get the latest)

Returns

bool True if the transform is possible, false otherwise.

static from_reference_frame(*reference_frame*)
Construct Transformer instance from static reference frame tree.

Parameters

reference_frame [ReferenceFrame] Reference frame instance from which to construct the transformer.

Returns

Transformer Transformer instance.

lookup_transform(*target_frame*, *source_frame*, *time=0.0*)
Get the transform from the source frame to the target frame.

Parameters

target_frame [str] Name of the frame to transform into.

source_frame [str] Name of the input frame.

time [float, default 0.0] Time at which to get the transform. (0 will get the latest)

Returns

t [tuple, len 3] The translation between the frames.

r [tuple, len 4] The rotation between the frames.

`set_transform_static`(*reference_frame*)
Add static transform from reference frame to buffer.

Parameters

reference_frame [ReferenceFrame] Static reference frame to add.

`set_transforms`(*reference_frame*)
Add transforms from moving reference frame to buffer.

Parameters

reference_frame [ReferenceFrame] Static reference frame to add.

`transform_point`(*p*, *target_frame*, *source_frame*, *time=0.0*)
Transform a point from the source frame to the target frame.

Parameters

p [iterable, len 3] Input point in source frame.

target_frame [str] Name of the frame to transform into.

source_frame [str] Name of the input frame.

time [float, default 0.0] Time at which to get the transform. (0 will get the latest)

Returns

tuple, len 3 Transformed point in target frame.

`transform_pose`(*p*, *o*, *target_frame*, *source_frame*, *time=0.0*)
Transform a pose from the source frame to the target frame.

Parameters

p [iterable, len 3] Input position in source frame.

o [iterable, len 3] Input orientation in source frame.

target_frame [str] Name of the frame to transform into.

source_frame [str] Name of the input frame.

time [float, default 0.0] Time at which to get the transform. (0 will get the latest)

Returns

pt [tuple, len 3] Transformed position in target frame.

ot [tuple, len 4] Transformed orientation in target frame.

`transform_quaternion`(*q*, *target_frame*, *source_frame*, *time=0.0*)
Transform a quaternion from the source frame to the target frame.

Parameters

q [iterable, len 4] Input quaternion in source frame.

target_frame [str] Name of the frame to transform into.

source_frame [str] Name of the input frame.

time [float, default 0.0] Time at which to get the transform. (0 will get the latest)

Returns

tuple, len 4 Transformed quaternion in target med quaternion in target frame.

transform_vector(*v*, *target_frame*, *source_frame*, *time=0.0*)
Transform a vector from the source frame to the target frame.

Parameters

v [iterable, len 3] Input vector in source frame.

target_frame [str] Name of the frame to transform into.

source_frame [str] Name of the input frame.

time [float, default 0.0] Time at which to get the transform. (0 will get the latest)

Returns

tuple, len 3 Transformed vector in target frame.

init_node	Register a client node with the master.
play_publisher	Interactive widget for playing back messages from a publisher.

rigid_body_motion.ros.init_node

rigid_body_motion.ros.**init_node**(*name*, *start_master=False*)
Register a client node with the master.

Parameters

name: str Name of the node.

start_master: bool, default False If True, start a ROS master if one isn't already running.

Returns

master: ROSLaunchParent or ROSMasterStub instance If a ROS master was started by this method, returns a `ROSLaunchParent` instance that can be used to shut down the master with its `shutdown()` method. Otherwise, a `ROSMasterStub` is returned that shows a warning when its `shutdown()` method is called.

rigid_body_motion.ros.play_publisher

rigid_body_motion.ros.**play_publisher**(*publisher*, *step=1*, *speed=1.0*, *skip=None*, *timestamps=None*)
Interactive widget for playing back messages from a publisher.

Parameters

publisher: object Any object with a `publish` method that accepts an `idx` parameter and publishes a message corresponding to that index.

step: int, default 1 Difference in indexes between consecutive messages, e.g. if `step=2` every second message will be published.

speed: float, default 1.0 Playback speed.

skip: int, optional Number of messages to skip with the forward and backward buttons.

timestamps: array_like, datetime64 dtype, optional Timestamps of publisher messages that determine time difference between messages and total number of messages. The time difference is calculated as the mean difference between the timestamps, i.e. it assumes that the

timestamps are more or less regular. If not provided, the publisher must have a `timestamps` attribute which will be used instead.

1.7.5 xarray Accessors

DataArray.rbm.qinterp	Quaternion interpolation.
DataArray.rbm.qinv	Quaternion inverse.

xarray.DataArray.rbm.qinterp

DataArray.rbm.**qinterp**(*coords=None*, *qdim='quaternion_axis'*, ***coords_kwargs*)

Quaternion interpolation.

Parameters

coords: dict, optional Mapping from dimension names to the new coordinates. New coordinate can be a scalar, array-like or DataArray.

qdim: str, default "quaternion_axis" Name of the dimension representing the quaternions.

****coords_kwargs** [{dim: coordinate, ...}, optional] The keyword arguments form of `coords`. One of coords or coords_kwargs must be provided.

Returns

interpolated: xr.DataArray New array on the new coordinates.

Examples

```
>>> import xarray as xr
>>> import rigid_body_motion as rbm
>>> ds_head = xr.load_dataset(rbm.example_data["head"])
>>> ds_left_eye = xr.load_dataset(rbm.example_data["left_eye"])
>>> ds_head.orientation.rbm.qinterp(time=ds_left_eye.time)
<xarray.DataArray 'orientation' (time: 113373, quaternion_axis: 4)>
array(...)
Coordinates:
  * time             (time) datetime64[ns] ...
  * quaternion_axis  (quaternion_axis) object 'w' 'x' 'y' 'z'
Attributes:
    long_name:  Orientation
```

xarray.DataArray.rbm.qinv

DataArray.rbm.**qinv**(*qdim='quaternion_axis'*)

Quaternion inverse.

Parameters

qdim: str, default "quaternion_axis" Name of the dimension representing the quaternions.

Returns

inverse: xr.DataArray New array with inverted quaternions.

Examples

```
>>> import xarray as xr
>>> import rigid_body_motion as rbm
>>> ds_head = xr.load_dataset(rbm.example_data["head"])
>>> ds_head.orientation.rbm.qinv()
<xarray.DataArray 'orientation' (time: 66629, quaternion_axis: 4)>
array(...)
Coordinates:
  * time             (time) datetime64[ns] ...
  * quaternion_axis  (quaternion_axis) object 'w' 'x' 'y' 'z'
Attributes:
    long_name:  Orientation
```

1.7.6 Class member details

reference_frames

ReferenceFrame

Construction

ReferenceFrame.from_dataset	Construct a reference frame from a Dataset.
ReferenceFrame.from_translation_dataarray	Construct a reference frame from a translation DataArray.
ReferenceFrame.from_rotation_dataarray	Construct a reference frame from a rotation DataArray.
ReferenceFrame.from_rotation_matrix	Construct a static reference frame from a rotation matrix.

rigid_body_motion.ReferenceFrame.from_dataset

classmethod ReferenceFrame.**from_dataset**(*ds*, *translation*, *rotation*, *timestamps*, *parent*, *name=None*, *inverse=False*, *discrete=False*)

Construct a reference frame from a Dataset.

Parameters

ds: xarray Dataset The dataset from which to construct the reference frame.

translation: str The name of the variable representing the translation wrt the parent frame.

rotation: str The name of the variable representing the rotation wrt the parent frame.

timestamps: str The name of the variable or coordinate representing the timestamps.

parent: str or ReferenceFrame The parent reference frame. If str, the frame will be looked up in the registry under that name.

name: str, default None The name of the reference frame.

inverse: bool, default False If True, invert the transform wrt the parent frame, i.e. the translation and rotation are specified for the parent frame wrt this frame.

discrete: bool, default False If True, transformations with timestamps are assumed to be events. Instead of interpolating between timestamps, transformations are fixed between their timestamp and the next one.

Returns

rf: ReferenceFrame The constructed reference frame.

rigid_body_motion.ReferenceFrame.from_translation_dataarray

classmethod ReferenceFrame.**from_translation_dataarray**(*da*, *timestamps*, *parent*, *name=None*, *inverse=False*, *discrete=False*)

Construct a reference frame from a translation DataArray.

Parameters

da: xarray DataArray The array that describes the translation of this frame wrt the parent frame.

timestamps: str The name of the variable or coordinate representing the timestamps.

parent: str or ReferenceFrame The parent reference frame. If str, the frame will be looked up in the registry under that name.

name: str, default None The name of the reference frame.

inverse: bool, default False If True, invert the transform wrt the parent frame, i.e. the translation is specified for the parent frame wrt this frame.

discrete: bool, default False If True, transformations with timestamps are assumed to be events. Instead of interpolating between timestamps, transformations are fixed between their timestamp and the next one.

Returns

rf: ReferenceFrame The constructed reference frame.

rigid_body_motion.ReferenceFrame.from_rotation_dataarray

classmethod ReferenceFrame.**from_rotation_dataarray**(*da*, *timestamps*, *parent*, *name=None*, *inverse=False*, *discrete=False*)

Construct a reference frame from a rotation DataArray.

Parameters

da: xarray DataArray The array that describes the rotation of this frame wrt the parent frame.

timestamps: str The name of the variable or coordinate representing the timestamps.

parent: str or ReferenceFrame The parent reference frame. If str, the frame will be looked up in the registry under that name.

name: str, default None The name of the reference frame.

inverse: bool, default False If True, invert the transform wrt the parent frame, i.e. the rotation is specified for the parent frame wrt this frame.

discrete: bool, default False If True, transformations with timestamps are assumed to be events. Instead of interpolating between timestamps, transformations are fixed between their timestamp and the next one.

Returns

rf: ReferenceFrame The constructed reference frame.

rigid_body_motion.ReferenceFrame.from_rotation_matrix

classmethod ReferenceFrame.**from_rotation_matrix**(*mat*, *parent*, *name=None*, *inverse=False*)

Construct a static reference frame from a rotation matrix.

Parameters

mat: array_like, shape (3, 3) The rotation matrix that describes the rotation of this frame wrt the parent frame.

parent: str or ReferenceFrame The parent reference frame. If str, the frame will be looked up in the registry under that name.

name: str, default None The name of the reference frame.

inverse: bool, default False If True, invert the transform wrt the parent frame, i.e. the rotation is specified for the parent frame wrt this frame.

Returns

rf: ReferenceFrame The constructed reference frame.

Transforms

ReferenceFrame.transform_points	Transform array of points from this frame to another.
ReferenceFrame.transform_quaternions	Transform array of quaternions from this frame to another.
ReferenceFrame.transform_vectors	Transform array of vectors from this frame to another.
ReferenceFrame.transform_angular_velocity	Transform array of angular velocities from this frame to another.
ReferenceFrame.transform_linear_velocity	Transform array of linear velocities from this frame to another.

rigid_body_motion.ReferenceFrame.transform_points

ReferenceFrame.**transform_points**(*arr*, *to_frame*, *axis=- 1*, *time_axis=0*, *timestamps=None*, *return_timestamps=False*)

Transform array of points from this frame to another.

Parameters

arr: array_like The array to transform.

to_frame: str or ReferenceFrame The target reference frame. If str, the frame will be looked up in the registry under that name.

axis: int, default -1 The axis of the array representing the spatial coordinates of the points.

time_axis: int, default 0 The axis of the array representing the timestamps of the points.

timestamps: array_like, optional The timestamps of the vectors, corresponding to the *time_axis* of the array. If not None, the axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like, shape (n_timestamps,) or None The timestamps after the transformation.

rigid_body_motion.ReferenceFrame.transform_quaternions

ReferenceFrame.**transform_quaternions**(*arr*, *to_frame*, *axis=- 1*, *time_axis=0*, *timestamps=None*, *return_timestamps=False*)

Transform array of quaternions from this frame to another.

Parameters

arr: array_like The array to transform.

to_frame: str or ReferenceFrame The target reference frame. If str, the frame will be looked up in the registry under that name.

axis: int, default -1 The axis of the array representing the spatial coordinates of the quaternions.

time_axis: int, default 0 The axis of the array representing the timestamps of the quaternions.

timestamps: array_like, optional The timestamps of the quaternions, corresponding to the *time_axis* of the array. If not None, the axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like, shape (n_timestamps,) or None The timestamps after the transformation.

rigid_body_motion.ReferenceFrame.transform_vectors

ReferenceFrame.**transform_vectors**(*arr*, *to_frame*, *axis=- 1*, *time_axis=0*, *timestamps=None*, *return_timestamps=False*)

Transform array of vectors from this frame to another.

Parameters

arr: array_like The array to transform.

to_frame: str or ReferenceFrame The target reference frame. If str, the frame will be looked up in the registry under that name.

axis: int, default -1 The axis of the array representing the spatial coordinates of the vectors.

time_axis: int, default 0 The axis of the array representing the timestamps of the vectors.

timestamps: array_like, optional The timestamps of the vectors, corresponding to the *time_axis* of the array. If not None, the axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

Returns

arr_transformed: array_like The transformed array.

ts: array_like, shape (n_timestamps,) or None The timestamps after the transformation.

rigid_body_motion.ReferenceFrame.transform_angular_velocity

ReferenceFrame.**transform_angular_velocity**(*arr*, *to_frame*, *what='reference_frame'*, *axis=- 1*, *time_axis=0*, *timestamps=None*, *return_timestamps=False*, *cutoff=None*)

Transform array of angular velocities from this frame to another.

Parameters

arr: array_like The array to transform.

to_frame: str or ReferenceFrame The target reference frame. If str, the frame will be looked up in the registry under that name.

what: str, default "reference_frame" What frame of the velocity to transform. Can be "reference_frame", "moving_frame" or "representation_frame".

axis: int, default -1 The axis of the array representing the spatial coordinates of the velocities.

time_axis: int, default 0 The axis of the array representing the timestamps of the velocities.

timestamps: array_like, optional The timestamps of the velocities, corresponding to the *time_axis* of the array. If not None, the axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the twist estimation as a fraction of the Nyquist frequency.

Returns

arr_transformed: array_like The transformed array.

ts: array_like, shape (n_timestamps,) or None The timestamps after the transformation.

See also:

transform_angular_velocity

rigid_body_motion.ReferenceFrame.transform_linear_velocity

ReferenceFrame.**transform_linear_velocity**(*arr*, *to_frame*, *what='reference_frame'*, *moving_frame=None*, *reference_frame=None*, *axis=- 1*, *time_axis=0*, *timestamps=None*, *return_timestamps=False*, *outlier_thresh=None*, *cutoff=None*)

Transform array of linear velocities from this frame to another.

Parameters

arr: array_like The array to transform.

to_frame: str or ReferenceFrame The target reference frame. If str, the frame will be looked up in the registry under that name.

what: str, default "reference_frame" What frame of the velocity to transform. Can be "reference_frame", "moving_frame" or "representation_frame".

moving_frame: str or ReferenceFrame, optional The moving frame when transforming the reference frame of the velocity.

reference_frame: str or ReferenceFrame, optional The reference frame when transforming the moving frame of the velocity.

axis: int, default -1 The axis of the array representing the spatial coordinates of the velocities.

time_axis: int, default 0 The axis of the array representing the timestamps of the velocities.

timestamps: array_like, optional The timestamps of the velocities, corresponding to the *time_axis* of the array. If not None, the axis defined by *time_axis* will be re-sampled to the timestamps for which the transformation is defined.

return_timestamps: bool, default False If True, also return the timestamps after the transformation.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the twist estimation as a fraction of the Nyquist frequency.

outlier_thresh: float, optional Suppress outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values.

Returns

arr_transformed: array_like The transformed array.

ts: array_like, shape (n_timestamps,) or None The timestamps after the transformation.

See also:

transform_linear_velocity

Lookups

ReferenceFrame.lookup_transform	Look up the transformation from this frame to another.
ReferenceFrame.lookup_angular_velocity	Estimate angular velocity of this frame wrt a reference.
ReferenceFrame.lookup_linear_velocity	Estimate linear velocity of this frame wrt a reference.
ReferenceFrame.lookup_twist	Estimate linear and angular velocity of this frame wrt a reference.

rigid_body_motion.ReferenceFrame.lookup_transform

ReferenceFrame.**lookup_transform**(*to_frame*)

Look up the transformation from this frame to another.

Parameters

to_frame: str or ReferenceFrame The target reference frame. If str, the frame will be looked up in the registry under that name.

Returns

t: array_like, shape (3,) or (n_timestamps, 3) The translation from this frame to the target frame.

r: array_like, shape (4,) or (n_timestamps, 4) The rotation from this frame to the target frame.

ts: array_like, shape (n_timestamps,) or None The timestamps for which the transformation is defined.

See also:

lookup_transform

rigid_body_motion.ReferenceFrame.lookup_angular_velocity

ReferenceFrame.**lookup_angular_velocity**(*reference=None*, *represent_in=None*, *outlier_thresh=None*, *cutoff=None*, *mode='quaternion'*, *allow_static=False*, *return_timestamps=False*)

Estimate angular velocity of this frame wrt a reference.

Parameters

reference: str or ReferenceFrame, optional The reference frame wrt which the twist is estimated. Defaults to the parent frame.

represent_in: str or ReferenceFrame, optional The reference frame in which the twist is represented. Defaults to the parent frame.

outlier_thresh: float, optional Suppress samples where the norm of the second-order differences of the rotation is above *outlier_thresh* and interpolate the missing values.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the estimation as a fraction of the Nyquist frequency.

mode: str, default "quaternion" If "quaternion", compute the angular velocity from the quaternion derivative. If "rotation_vector", compute the angular velocity from the gradient of the axis-angle representation of the rotations.

allow_static: bool, default False If True, return a zero velocity vector and None for timestamps if the transform between this frame and the reference frame is static. Otherwise, a *ValueError* will be raised.

return_timestamps: bool, default False If True, also return the timestamps of the lookup.

Returns

angular: numpy.ndarray, shape (N, 3) Angular velocity of moving frame wrt reference frame, represented in representation frame.

timestamps: each numpy.ndarray Timestamps of the angular velocity.

rigid_body_motion.ReferenceFrame.lookup_linear_velocity

ReferenceFrame.**lookup_linear_velocity**(*reference=None*, *represent_in=None*, *outlier_thresh=None*, *cutoff=None*, *allow_static=False*, *return_timestamps=False*)

Estimate linear velocity of this frame wrt a reference.

Parameters

reference: str or ReferenceFrame, optional The reference frame wrt which the twist is estimated. Defaults to the parent frame.

represent_in: str or ReferenceFrame, optional The reference frame in which the twist is represented. Defaults to the parent frame.

outlier_thresh: float, optional Suppress outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the estimation as a fraction of the Nyquist frequency.

allow_static: bool, default False If True, return a zero velocity vector and None for timestamps if the transform between this frame and the reference frame is static. Otherwise, a *ValueError* will be raised.

return_timestamps: bool, default False If True, also return the timestamps of the lookup.

Returns

linear: numpy.ndarray, shape (N, 3) Linear velocity of moving frame wrt reference frame, represented in representation frame.

timestamps: each numpy.ndarray Timestamps of the linear velocity.

rigid_body_motion.ReferenceFrame.lookup_twist

ReferenceFrame.**lookup_twist**(*reference=None*, *represent_in=None*, *outlier_thresh=None*, *cutoff=None*, *mode='quaternion'*, *allow_static=False*, *return_timestamps=False*)

Estimate linear and angular velocity of this frame wrt a reference.

Parameters

reference: str or ReferenceFrame, optional The reference frame wrt which the twist is estimated. Defaults to the parent frame.

represent_in: str or ReferenceFrame, optional The reference frame in which the twist is represented. Defaults to the parent frame.

outlier_thresh: float, optional Suppress outliers by throwing out samples where the norm of the second-order differences of the position is above *outlier_thresh* and interpolating the missing values.

cutoff: float, optional Frequency of a low-pass filter applied to linear and angular velocity after the estimation as a fraction of the Nyquist frequency.

mode: str, default "quaternion" If "quaternion", compute the angular velocity from the quaternion derivative. If "rotation_vector", compute the angular velocity from the gradient of the axis-angle representation of the rotations.

allow_static: bool, default False If True, return a zero velocity vector and None for timestamps if the transform between this frame and the reference frame is static. Otherwise, a *ValueError* will be raised.

return_timestamps: bool, default False If True, also return the timestamps of the lookup.

Returns

linear: numpy.ndarray, shape (N, 3) Linear velocity of moving frame wrt reference frame, represented in representation frame.

angular: numpy.ndarray, shape (N, 3) Angular velocity of moving frame wrt reference frame, represented in representation frame.

timestamps: each numpy.ndarray Timestamps of the twist.

Registry

ReferenceFrame.register	Register this frame in the registry.
ReferenceFrame.deregister	Remove this frame from the registry.

rigid_body_motion.ReferenceFrame.register

ReferenceFrame.**register**(*update=False*)
Register this frame in the registry.

Parameters

update: bool, default False If True, overwrite if there is a frame with the same name in the registry.

rigid_body_motion.ReferenceFrame.deregister

ReferenceFrame.**deregister**()
Remove this frame from the registry.

1.8 Development roadmap

The vision for this project is to provide a universal library for analysis of recorded motion. Some of the categories where a lot of features are still to be implemented are detailed below.

1.8.1 IO functions

Import routines for a variety of data formats from common motion capture and IMU systems.

1.8.2 Estimators

Common transform estimators such as iterative closest points (ICP).

1.8.3 Pandas support

The same metadata-aware mechanism for pandas as for xarray.

1.8.4 ROS integration

Leverage tools provided by ROS to supplement functionality of the library. One example would be 3D plotting with RViz. ROS has recently made its way to conda forge, and packages such as jupyter-ros facilitate integration with modern Python workflows.

1.8.5 Units

Support for unit conversions and unit-aware transforms. Possibly leveraging packages such as pint or unit support in xarray.

1.9 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.9.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/phausamann/rigid-body-motion/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

rigid-body-motion could always use more documentation, whether as part of the official rigid-body-motion docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/phausamann/rigid-body-motion/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.9.2 Get Started!

Ready to contribute? Here's how to set up *rigid-body-motion* for local development.

1. Fork the *rigid-body-motion* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/rigid-body-motion.git
```

3. The recommended way of setting up the project is in a conda environment:

```
$ conda env create
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests:

```
$ conda activate rbm
$ pytest
```

6. The project uses black for code formatting, isort for import sorting and flake8 for linting. You can set up these checks as pre-commit hooks:

```
$ conda activate rbm
$ conda install pre-commit
$ pre-commit install
```

   Note that the project currently uses black version 19.10b0.

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

1.9.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in HISTORY.rst.
3. The pull request should work for Python 3.6, 3.7, 3.8 and 3.9. See the checks at the bottom of the pull request page to resolve any issues.

1.9.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
```

1.10 Credits

1.10.1 Development Lead

- Peter Hausamann <peter@hausamann.de>

1.10.2 Contributors

None yet. Why not be the first?

1.11 History

1.11.1 0.9.2 (TBD)

Bug fixes & improvements

- Fixed an error related to a deprecated matplotlib.pyplot reference, see *Issue #35 <https://github.com/phausamann/rigid-body-motion/issues/35>*.

1.11.2 0.9.1 (January 13th, 2022)

Bug fixes & improvements

- Fixed package installation through pip (version 0.9.0 is no longer available).

1.11.3 0.9.0 (December 29th, 2021)

Breaking changes

- Dropped support for Python 3.6.

Bug fixes & improvements

- Fixed issue with matplotlib versions >= 3.5.

1.11.4 0.8.0 (May 27th, 2021)

New features

- New `ros.init_node` method to initialize a ROS node and optionally start a ROS master.

Bug fixes & improvements

- All ROS dependencies are now lazily imported.

1.11.5 0.7.0 (May 19th, 2021)

New features

- New `from_euler_angles` utility method.

Bug fixes & improvements

- Importing ROS interface classes will not fail silently anymore and instead show the traceback of the import error.

1.11.6 0.6.0 (May 17th, 2021)

Breaking changes

- Example data is now fetched via the `pooch` library and no longer a part of the package itself.

New features

- New `io` module for import/export methods.
- New `ros.RosbagWriter` class for writing rosbag files.

1.11.7 0.5.0 (March 16th, 2021)

Breaking changes

- Top-level reference frame transform and lookup methods now all accept a `return_timestamps` argument that is `False` by default. Previously, methods would return timestamps only if the result of the transformation was timestamped. This does not affect the xarray interface.
- `lookup_transform` now returns the correct transformation from the base frame to the target frame (instead of the other way around).
- `ReferenceFrame.get_transformation` is deprecated and replaced by `ReferenceFrame.lookup_transform`.

New features

- New `plot` module with plotting methods for static reference frames and arrays of points, quaternions and vectors.
- New `lookup_pose` method that calculates the pose of a frame wrt another.

Bug fixes & improvements

- Fixed `"reference_frame"` attribute incorrectly set by `transform_vectors`.

1.11.8 0.4.1 (February 18th, 2021)

Bug fixes & improvements

- Fixed `transform_coordinates` failing when spatial dimension is first axis of array.
- Fixed `transform_linear_velocity` and `transform_angular_velocity` failing when reference frame or moving frame is transformed across only static transforms.
- Added `allow_static` parameter to `lookup_twist`, `lookup_angular_velocity` and `lookup_linear_velocity` to return zero velocity and no timestamps across only static transforms.

1.11.9 0.4.0 (February 11th, 2021)

New features

- New `lookup_linear_velocity` and `lookup_angular_velocity` top-level methods.
- New `render_tree` top-level method for printing out a graphical representation of a reference frame tree.
- `lookup_twist` now accepts a `mode` parameter to specify the mode for angular velocity calculation.

Bug fixes & improvements

- Fixed a bug where estimated angular velocity was all NaN when orientation contained NaNs.

1.11.10 0.3.0 (December 8th, 2020)

New features

- Reference frames with timestamps now accept the `discrete` parameter, allowing for transformations to be fixed from their timestamp into the future.
- `rbm` accessor for DataArrays implementing `qinterp` and `qinv` methods.
- New `best_fit_rotation` and `qinterp` top-level methods.

Bug fixes & improvements

- Refactor of internal timestamp matching mechanism defining a clear priority for target timestamps. This can result in slight changes of timestamps and arrays returned by transformations but will generally produce more accurate results.
- Added `mode` and `outlier_thresh` arguments to `estimate_angular_velocity`.
- Fixed issues with `iterative_closest_point`.

1.11.11 0.2.0 (October 22nd, 2020)

New features

- New `estimate_linear_velocity` and `estimate_angular_velocity` top-level methods.
- New `qmul` top-level method for multiplying quaternions.

1.11.12 0.1.2 (October 7th, 2020)

Improvements

- Use SQUAD instead of linear interpolation for quaternions.

1.11.13 0.1.1 (September 17th, 2020)

Bug fixes

- Fix transformations failing for DataArrays with non-numeric coords.

1.11.14 0.1.0 (September 17th, 2020)

- First release

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

B

C

D

E

F

G

H

I

L

P

Q

R

S

T

V

W