rowan Documentation

Release 1.3.0

Vyas Ramasubramani

Jan 06, 2021

Modules:

1 Getting Started **3**
- 1.1 Installation . . . 3
- 1.2 Quickstart . . . 3
- 1.3 Running Tests . . . 4
- 1.4 Running Benchmarks . . . 4
- 1.5 Building Documentation . . . 4

2 Support and Contribution **7**

3 Table of Contents **9**
- 3.1 rowan . . . 9
- 3.2 calculus . . . 20
- 3.3 geometry . . . 21
- 3.4 interpolate . . . 24
- 3.5 mapping . . . 26
- 3.6 random . . . 30
- 3.7 Development Guide . . . 31
- 3.8 License . . . 33
- 3.9 Changelog . . . 33
- 3.10 Credits . . . 38

4 Indices and tables **41**

Bibliography **43**

Python Module Index **45**

Index **47**

- *Getting Started*
  - *Installation*
  - *Quickstart*
  - *Running Tests*
  - *Running Benchmarks*
  - *Building Documentation*
- *Support and Contribution*
- *Table of Contents*
- *Indices and tables*

Welcome to the documentation for rowan, a package for working with quaternions! Quaternions, which form a number system with various interesting properties, were originally developed for classical mechanics. Although they have since been largely displaced from this application by vector mathematics, they have become a standard method of representing rotations in three dimensions. Quaternions are now commonly used for this purpose in various fields, including computer graphics and attitude control.

The package is built entirely on top of NumPy and represents quaternions using NumPy arrays, meaning that all functions support arbitrarily high-dimensional arrays of quaternions. Quaternions are encoded as arrays of shape `(..., 4)`, with the convention that the final dimension of an array ``(a, b, c, d)` represents the quaternion `a + bi + cj + dk`. This package provides tools for standard algebraic operations on quaternions as well as a number of additional tools for *e.g.* measuring distances between quaternions, interpolating between them, and performing basic point-cloud mapping. A particular focus of the rowan package is working with unit quaternions, which are a popular means of representing rotations in 3D. In order to provide a unified framework for working with the various rotation formalisms in 3D, rowan allows easy interconversion between these formalisms.

Core features of rowan include (but are not limited to):

- Algebra (multiplication, exponentiation, etc).
- Derivatives and integrals of quaternions.
- Rotation and reflection operations, with conversions to and from matrices, axis angles, etc.
- Various distance metrics for quaternions.
- Basic point set registration, including solutions of the Procrustes problem and the Iterative Closest Point algorithm.
- Quaternion interpolation (slerp, squad).

CHAPTER 1

Getting Started

1.1 Installation

The recommended methods for installing rowan are using **pip** or **conda**. To install the package from PyPI, execute:

```
$ pip install rowan --user
```

To install the package from conda, first add the **conda-forge** channel and then install rowan:

```
$ conda config --add channels conda-forge
$ conda install rowan
```

If you wish, you may also install rowan by cloning the repository and running the setup script:

```
$ git clone https://github.com/glotzerlab/rowan.git
$ cd rowan
$ python setup.py install --user
```

The minimum requirements for using rowan are:

- Python >= 3.6
- NumPy >= 1.15

1.2 Quickstart

This library can be used to work with quaternions by simply instantiating the appropriate NumPy arrays and passing them to the required functions. For example:

```
import rowan
import numpy as np
one = np.array([10, 0, 0, 0])
```

(continues on next page)

(continued from previous page)

```
one_unit = rowan.normalize(one)
assert(np.all(one_unit == np.array([1, 0, 0, 0])))
if not np.all(one_unit == rowan.multiply(one_unit, one_unit)):
    raise RuntimeError("Multiplication failed!")

one_vec = np.array([1, 0, 0])
rotated_vector = rowan.rotate(one_unit, one_vec)

mat = np.eye(3)
quat_rotate = rowan.from_matrix(mat)
alpha, beta, gamma = rowan.to_euler(quat_rotate)
quat_rotate_returned = rowan.from_euler(alpha, beta, gamma)
identity = rowan.to_matrix(quat_rotate_returned)
```

1.3 Running Tests

The package is currently tested for Python >= 3.6 on Unix-like systems. Continuous integrated testing is performed using CircleCI on these Python versions with NumPy versions 1.15 and above.

To run the packaged unit tests, execute the following line from the root of the repository:

```
python -m unittest discover tests
```

To check test coverage, make sure the coverage module is installed:

```
pip install coverage
```

and then run the packaged unit tests with the coverage module:

```
coverage run -m unittest discover tests
```

1.4 Running Benchmarks

Benchmarks for the package are contained in a Jupyter notebook in the `benchmarks` folder in the root of the repository. If you do not have or do not wish to use the notebook format, an equivalent Benchmarks.py script is also included. The benchmarks compare rowan to two alternative packages, so you will need to install `pyquaternion` and `numpy_quaternion` if you wish to see those comparisons.

1.5 Building Documentation

You can also build this documentation from source if you clone the repository. The documentation is written in reStructuredText and compiled using Sphinx. To build from source, first install Sphinx:

```
pip install sphinx sphinx_rtd_theme
```

You can then use Sphinx to create the actual documentation in either PDF or HTML form by running the following commands in the rowan root directory:

```
cd doc
make html # For html output
make latexpdf # For a LaTeX compiled PDF file
open build/html/index.html
```

CHAPTER 2

Support and Contribution

This package is hosted on GitHub. Please report any bugs or problems that you find on the issue tracker.

All contributions to rowan are welcomed via pull requests! Please see the development guide for more information on requirements for new code.

CHAPTER 3

Table of Contents

3.1 rowan

Overview

rowan.conjugate	Conjugates an array of quaternions.
rowan.inverse	Compute the inverse of an array of quaternions.
rowan.exp	Compute the natural exponential function e^q.
rowan.expb	Compute the exponential function b^q.
rowan.exp10	Compute the exponential function 10^q.
rowan.log	Compute the quaternion natural logarithm.
rowan.logb	Compute the quaternion logarithm to some base b.
rowan.log10	Compute the quaternion logarithm base 10.
rowan.multiply	Multiplies two arrays of quaternions.
rowan.divide	Divides two arrays of quaternions.
rowan.norm	Compute the quaternion norm.
rowan.normalize	Normalize quaternions.
rowan.rotate	Rotate a list of vectors by a corresponding set of quaternions.
rowan.vector_vector_rotation	Find the quaternion to rotate one vector onto another.
rowan.from_euler	Convert Euler angles to quaternions.
rowan.to_euler	Convert quaternions to Euler angles.
rowan.from_matrix	Convert the rotation matrices mat to quaternions.
rowan.to_matrix	Convert quaternions into rotation matrices.
rowan.from_axis_angle	Find quaternions to rotate a specified angle about a specified axis.
rowan.to_axis_angle	Convert the quaternions in q to axis-angle representations.
rowan.from_mirror_plane	Generate quaternions from mirror plane equations.

Continued on next page

Table 1 – continued from previous page

rowan.reflect	Reflect a list of vectors by a corresponding set of quaternions.
rowan.equal	Check whether two sets of quaternions are equal.
rowan.not_equal	Check whether two sets of quaternions are not equal.
rowan.isfinite	Test element-wise for finite quaternions.
rowan.isinf	Test element-wise for infinite quaternions.
rowan.isnan	Test element-wise for NaN quaternions.

Details

The rowan package for working with quaternions.

The core *rowan* package contains functions for operating on quaternions. The core package is focused on robust implementations of key functions like multiplication, exponentiation, norms, and others. Simple functionality such as addition is inherited directly from NumPy due to the representation of quaternions as NumPy arrays. Many core NumPy functions implemented for normal arrays are reimplemented to work on quaternions (such as *allclose()* and *isfinite()*). Additionally, NumPy broadcasting is enabled throughout rowan unless otherwise specified. This means that any function of 2 (or more) quaternions can take arrays of shapes that do not match and return results according to NumPy's broadcasting rules.

rowan.**allclose**(*p*, *q*, ***kwargs*)

Check whether two sets of quaternions are all close.

This is a direct wrapper of the corresponding NumPy function.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.
- ****kwargs** – Keyword arguments to pass to np.allclose.

Returns Whether all of p and q are close.

Return type bool

Example:

```
>>> rowan.allclose([1, 0, 0, 0], [1, 0, 0, 0])
True
```

rowan.**conjugate**(*q*)

Conjugates an array of quaternions.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Conjugates of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.conjugate([0.5, 0.5, -0.5, 0.5])
array([ 0.5, -0.5,  0.5, -0.5])
```

rowan.**divide**(*qi*, *qj*)

Divides two arrays of quaternions.

Division is non-commutative; this function returns $q_i q_j^{-1}$.

Parameters

- **qi** ((..., 4) numpy.ndarray) – Dividend quaternions.
- **qj** ((..., 4) numpy.ndarray) – Divisor quaternions.

Returns Element-wise quotients of q (obeying broadcasting rules up to the last dimension of qi and qj).

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.divide([1, 0, 0, 0], [2, 0, 0, 0])
array([0.5, 0. , 0. , 0. ])
```

rowan.**exp**(q)

Compute the natural exponential function e^q.

The exponential of a quaternion in terms of its scalar and vector parts $q = a + \boldsymbol{v}$ is defined by exponential power series: formula $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$ as follows:

$$e^q = e^{a+v} \tag{3.1}$$

$$= e^a \left(\sum_{k=0}^{\infty} \frac{v^k}{k!} \right) \tag{3.2}$$

$$= e^a \left(\cos||\boldsymbol{v}|| + \frac{\boldsymbol{v}}{||\boldsymbol{v}||} \sin||\boldsymbol{v}|| \right) \tag{3.3}$$

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Exponentials of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.exp([1, 0, 0, 0])
array([2.71828183, 0.        , 0.        , 0.        ])
```

rowan.**expb**(q, b)

Compute the exponential function b^q.

We define the exponential of a quaternion to an arbitrary base relative to the exponential function e^q using the change of base formula as follows:

$$b^q = y \tag{3.4}$$

$$q = \log_b y = \frac{\ln y}{\ln b} \tag{3.5}$$

$$y = e^{q \ln b} \tag{3.6}$$

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Exponentials of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.expb([1, 0, 0, 0], 2)
array([2., 0., 0., 0.])
```

rowan.**exp10**(*q*)

Compute the exponential function 10^q.

Wrapper around *expb()*.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Exponentials of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.exp10([1, 0, 0, 0])
array([10.,  0.,  0.,  0.])
```

rowan.**equal**(*p*, *q*)

Check whether two sets of quaternions are equal.

This function is a simple wrapper that checks array equality and then aggregates along the quaternion axis.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.

Returns Whether p and q are equal.

Return type (...) numpy.ndarray of bool

Example:

```
>>> rowan.equal([1, 0, 0, 0], [1, 0, 0, 0])
True
```

rowan.**from_axis_angle**(*axes*, *angles*)

Find quaternions to rotate a specified angle about a specified axis.

All angles are assumed to be **counterclockwise** rotations about the axis.

Parameters

- **axes** ((..., 3) numpy.ndarray) – An array of vectors (the axes).
- **angles** (float or (..., 1) numpy.ndarray) – An array of angles in radians. Will be broadcasted to match shape of axes as needed.

Returns The corresponding rotation quaternions.

Return type (..., 4) numpy.ndarray

Example:

```
>>> import numpy as np
>>> rowan.from_axis_angle([[1, 0, 0]], np.pi/3)
array([[0.8660254, 0.5      , 0.       , 0.       ]])
```

rowan.**from_euler**(*alpha*, *beta*, *gamma*, *convention='zyx'*, *axis_type='intrinsic'*)

Convert Euler angles to quaternions.

For generality, the rotations are computed by composing a sequence of quaternions corresponding to axis-angle rotations. While more efficient implementations are possible, this method was chosen to prioritize flexibility since it works for essentially arbitrary Euler angles as long as intrinsic and extrinsic rotations are not intermixed.

Parameters

- **alpha** ((...) numpy.ndarray) – Array of α values in radians.
- **beta** ((...) numpy.ndarray) – Array of β values in radians.
- **gamma** ((...) numpy.ndarray) – Array of γ values in radians.
- **convention** (*str*) – One of the 12 valid conventions xzx, xyx, yxy, yzy, zyz, zxz, xzy, xyz, yxz, yzx, zyx, zxy.
- **axes** (*str*) – Whether to use extrinsic or intrinsic rotations.

Returns Quaternions corresponding to the input angles.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.from_euler(0.3, 0.5, 0.7)
array([0.91262714, 0.29377717, 0.27944389, 0.05213241])
```

rowan.**from_matrix**(*mat*, *require_orthogonal=True*)

Convert the rotation matrices mat to quaternions.

This method uses the algorithm described by Bar-Itzhack in [Itzhack00]. The idea is to construct a matrix K whose largest eigenvalue corresponds to the desired quaternion. One of the strengths of the algorithm is that for nonorthogonal matrices it gives the closest quaternion representation rather than failing outright.

Parameters **mat** ((..., 3, 3) numpy.ndarray) – An array of rotation matrices.

Returns The corresponding rotation quaternions.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.from_matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
array([ 1., -0., -0., -0.])
```

rowan.**from_mirror_plane**(*x*, *y*, *z*)

Generate quaternions from mirror plane equations.

Reflection quaternions can be constructed from the form $(0, x, y, z)$, *i.e.* with zero real component. The vector (x, y, z) is the normal to the mirror plane.

Parameters

- **x** ((...) numpy.ndarray) – First planar component.
- **y** ((...) numpy.ndarray) – Second planar component.
- **z** ((...) numpy.ndarray) – Third planar component.

Returns Quaternions reflecting about the input plane (x, y, z).

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.from_mirror_plane(*(1, 2, 3))
array([0., 1., 2., 3.])
```

rowan.**inverse**(*q*)

Compute the inverse of an array of quaternions.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Inverses of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.inverse([1, 0, 0, 0])
array([ 1., -0., -0., -0.])
```

rowan.**isclose**(*p*, *q*, ***kwargs*)

Element-wise check of whether two sets of quaternions are close.

This function is a simple wrapper that checks using the corresponding NumPy function and then aggregates along the quaternion axis.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.
- ****kwargs** – Keyword arguments to pass to np.isclose.

Returns Whether p and q are close element-wise.

Return type (...) numpy.ndarray of bool

Example:

```
>>> rowan.isclose([[1, 0, 0, 0]], [[1, 0, 0, 0]])
array([ True])
```

rowan.**isinf**(*q*)

Test element-wise for infinite quaternions.

A quaternion is defined as infinite if any elements are infinite.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions

Returns Whether q is infinite.

Return type (...) numpy.ndarray of bool

Example:

```
>>> import numpy as np
>>> rowan.isinf([np.nan, 0, 0, 0])
False
```

rowan.**isfinite**(*q*)

Test element-wise for finite quaternions.

A quaternion is defined as finite if all elements are finite.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Whether q is finite.

Return type (...) numpy.ndarray of bool

Example:

```
>>> rowan.isfinite([1, 0, 0, 0])
True
```

rowan.**isnan**(*q*)

Test element-wise for NaN quaternions.

A quaternion is defined as NaN if any elements are NaN.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Whether q is NaN.

Return type (...) numpy.ndarray of bool

Example:

```
>>> import numpy as np
>>> rowan.isnan([np.nan, 0, 0, 0])
True
```

rowan.**is_unit**(*q*)

Check if all input quaternions have unit norm.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Whether or not all inputs are unit quaternions.

Return type (...) numpy.ndarray of bool

Example:

```
>>> rowan.is_unit([10, 0, 0, 0])
False
```

rowan.**log**(*q*)

Compute the quaternion natural logarithm.

The natural of a quaternion in terms of its scalar and vector parts $q = a + \boldsymbol{v}$ is defined by inverting the exponential formula (see *exp()*), and is defined by the formula $\frac{x^k}{k!}$ as follows:

$$\ln(q) = \ln||q|| + \frac{\boldsymbol{v}}{||\boldsymbol{v}||} \arccos\left(\frac{a}{q}\right) \quad (3.7)$$

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Logarithms of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.log([1, 0, 0, 0])
array([0., 0., 0., 0.])
```

rowan.**logb**(*q*, *b*)

Compute the quaternion logarithm to some base b.

The quaternion logarithm for arbitrary bases is defined using the standard change of basis formula relative to the natural logarithm.

$$\begin{aligned} \log_b q &= y \\ q &= b^y \\ \ln q &= y \ln b \\ y &= \log_b q = \frac{\ln q}{\ln b} \end{aligned} \tag{3.8}$$

Parameters

- **q** ((..., 4) numpy.ndarray) – Array of quaternions.
- **n** ((...) numpy.ndarray) – Scalars to use as log bases.

Returns Logarithms of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.logb([1, 0, 0, 0], 2)
array([0., 0., 0., 0.])
```

rowan.**log10**(*q*)

Compute the quaternion logarithm base 10.

Wrapper around *logb()*.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Logarithms of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.log10([1, 0, 0, 0])
array([0., 0., 0., 0.])
```

rowan.**multiply**(*qi*, *qj*)

Multiplies two arrays of quaternions.

Note that quaternion multiplication is generally non-commutative, so the first and second set of quaternions must be passed in the correct order.

Parameters

- **qi** ((..., 4) numpy.ndarray) – Array of left quaternions.
- **qj** ((..., 4) numpy.ndarray) – Array of right quaternions.

Returns Element-wise products of q (obeying broadcasting rules up to the last dimension of qi and qj).

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.multiply([1, 0, 0, 0], [2, 0, 0, 0])
array([2., 0., 0., 0.])
```

rowan.**norm**(*q*)

Compute the quaternion norm.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Norms of q.

Return type (...) numpy.ndarray

Example:

```
>>> rowan.norm([10, 0, 0, 0])
10.0
```

rowan.**normalize**(*q*)

Normalize quaternions.

Parameters **q** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Normalized versions of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.normalize([10, 0, 0, 0])
array([1., 0., 0., 0.])
```

rowan.**not_equal**(*p*, *q*)

Check whether two sets of quaternions are not equal.

This function is a simple wrapper that checks array equality and then aggregates along the quaternion axis.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.

Returns Whether p and q are unequal.

Return type (...) numpy.ndarray of bool

Example:

```
>>> rowan.not_equal([-1, 0, 0, 0], [1, 0, 0, 0])
True
```

rowan.**power**(*q*, *n*)

Compute the power of a quaternion q^n.

Quaternions raised to a scalar power are defined according to the polar decomposition angle θ and vector $\hat{u}$: $q^n = ||q||^n (\cos(n\theta) + \hat{u}\sin(n\theta))$. However, this can be computed more efficiently by noting that $q^n = \exp(n \ln(q))$.

Parameters

- **q** ((..., 4) numpy.ndarray) – Array of quaternions.
- **n** ((...) numpy.ndarray) – Scalars to exponentiate quaternions with.

Returns Powers of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.power([1, 0, 0, 0], 5)
array([1., 0., 0., 0.])
```

rowan.**reflect**(*q*, *v*)

Reflect a list of vectors by a corresponding set of quaternions.

For help constructing a mirror plane, see *from_mirror_plane()*.

Parameters

- **q** ((..., 4) numpy.ndarray) – Array of quaternions.
- **v** ((..., 3) numpy.ndarray) – Array of vectors.

Returns The result of reflecting v using q.

Return type (..., 3) numpy.ndarray

Example:

```
>>> rowan.reflect([1, 0, 0, 0], [1, 1, 1])
array([1., 1., 1.])
```

rowan.**rotate**(*q*, *v*)

Rotate a list of vectors by a corresponding set of quaternions.

Parameters

- **q** ((..., 4) numpy.ndarray) – Array of quaternions.
- **v** ((..., 3) numpy.ndarray) – Array of vectors.

Returns The result of rotating v using q.

Return type (..., 3) numpy.ndarray

Example:

```
>>> rowan.rotate([1, 0, 0, 0], [1, 1, 1])
array([1., 1., 1.])
```

rowan.**to_axis_angle**(*q*)

Convert the quaternions in q to axis-angle representations.

The output angles are **counterclockwise** rotations about the axis.

Parameters **q** ((..., 4) numpy.ndarray) – An array of quaternions.

Returns The axes and the angles (in radians).

Return type tuple[(..., 3) numpy.ndarray, (...) numpy.ndarray]

Example:

```
>>> rowan.to_axis_angle([[1, 0, 0, 0]])
(array([[0., 0., 0.]]), array([0.]))
```

rowan.**to_euler**(*q*, *convention='zyx'*, *axis_type='intrinsic'*)

Convert quaternions to Euler angles.

Euler angles are returned in the sequence provided, so in, *e.g.*, the default case ('zyx'), the angles returned are for a rotation $Z(\alpha)Y(\beta)X(\gamma)$.

Note: In all cases, the α and γ angles are between $\pm\pi$. For proper Euler angles, β is between 0 and π degrees. For Tait-Bryan angles, β lies between $\pm\pi/2$.

For simplicity, quaternions are converted to matrices, which are then converted to their Euler angle representations. All equations for rotations are derived by considering compositions of the three elemental rotations about the three Cartesian axes.

Extrinsic rotations are represented by matrix multiplications in the proper order, so $z - y - x$ is represented by the multiplication XYZ so that the system is rotated first about Z, then about Y, then finally X. For intrinsic rotations, the order of rotations is reversed, meaning that it matches the order in which the matrices actually appear *i.e.* the $z - y' - x''$ convention (yaw, pitch, roll) corresponds to the multiplication of matrices ZYX. For proof of the relationship between intrinsic and extrinsic rotations, see the Wikipedia page on Davenport chained rotations.

For more information, see the Wikipedia page for Euler angles (specifically the section on converting between representations).

Warning: Euler angles are a highly problematic representation for a number of reasons, not least of which is the large number of possible conventions and their relative imprecision when compared to using quaternions (or axis-angle representations). If possible, you should avoid Euler angles and work with quaternions instead. If Euler angles are required, note that they are susceptible to gimbal lock, which leads to ambiguity in the representation of a given rotation. To address this issue, in cases where gimbal lock arises, *to_euler()* adopts the convention that $\gamma = 0$ and represents the rotation entirely in terms of β and α.

Parameters

- **q** ((..., 4) numpy.ndarray) – Quaternions to transform.
- **convention** (*str*) – One of the 6 valid conventions zxz, xyx, yzy, zyz, xzx, yxy.
- **axes** (*str*) – Whether to use extrinsic or intrinsic.

Returns Euler angles (α, β, γ) corresponding to q.

Return type (..., 3) numpy.ndarray

Example:

```
>>> import numpy as np
>>> rands = np.random.rand(100, 3)
>>> alpha, beta, gamma = rands.T
>>> ql = rowan.from_euler(alpha, beta, gamma)
>>> alpha_return, beta_return, gamma_return = np.split(
...     rowan.to_euler(ql), 3, axis = 1)
>>> assert(np.allclose(alpha_return.flatten(), alpha))
>>> assert(np.allclose(beta_return.flatten(), beta))
>>> assert(np.allclose(gamma_return.flatten(), gamma))
```

rowan.**to_matrix**(*q*, *require_unit=True*)

Convert quaternions into rotation matrices.

Uses the conversion described on Wikipedia.

Parameters **q** ((..., 4) numpy.ndarray) – An array of quaternions.

Returns The corresponding rotation matrices.

Return type (..., 3, 3) numpy.ndarray

Example:

```
>>> rowan.to_matrix([1, 0, 0, 0])
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

rowan.**vector_vector_rotation**(*v1*, *v2*)

Find the quaternion to rotate one vector onto another.

Note: Vector-vector rotation is underspecified, with one degree of freedom possible in the resulting quaternion. This method chooses to rotate by π around the vector bisecting v1 and v2.

Parameters

- **v1** ((..., 3) numpy.ndarray) – Array of vectors to rotate.
- **v2** ((..., 3) numpy.ndarray) – Array of vector to rotate onto.

Returns Quaternions that rotate `v1` onto `v2`.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.vector_vector_rotation([1, 0, 0], [0, 1, 0])
array([6.12323400e-17, 7.07106781e-01, 7.07106781e-01, 0.00000000e+00])
```

3.2 calculus

Overview

rowan.calculus.derivative	Compute the instantaneous derivative of unit quaternions.
rowan.calculus.integrate	Integrate unit quaternions by angular velocity.

Details

Compute derivatives and integrals of quaternions.

rowan.calculus.**derivative**(*q*, *v*)

Compute the instantaneous derivative of unit quaternions.

Derivatives of quaternions are defined by the equation:

$$\dot{q} = \frac{1}{2}\boldsymbol{v}q$$

A derivation is provided here. For a more thorough explanation, see this page.

Parameters

- **q** ((..., 4) numpy.ndarray) – Array of quaternions.
- **v** ((..., 3) numpy.ndarray) – Array of angular velocities.

Returns Derivatives of `q`.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.calculus.derivative([1, 0, 0, 0], [1, 0, 0])
array([0. , 0.5, 0. , 0. ])
```

rowan.calculus.**integrate**(*q*, *v*, *dt*)

Integrate unit quaternions by angular velocity.

The integral uses the following equation:

$$\dot{q} = \exp\left(\frac{1}{2}\boldsymbol{v}dt\right) q$$

Note that this formula uses the quaternion exponential, so the argument to the exponential (which appears to be a vector) is promoted to a quaternion with scalar part 0 before the exponential is taken. A concise derivation is provided in this paper. This webpage contains a more thorough explanation.

Parameters

- **q** ((..., 4) numpy.ndarray) – Array of quaternions.
- **v** ((..., 3) numpy.ndarray) – Array of angular velocities.
- **dt** ((...) numpy.ndarray) – Array of timesteps.

Returns Integrals of q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.calculus.integrate([1, 0, 0, 0], [0, 0, 1e-2], 1)
array([0.9999875 , 0.        , 0.        , 0.00499998])
```

3.3 geometry

Overview

rowan.geometry.distance	Determine the distance between quaternions p and q.
rowan.geometry.sym_distance	Determine the distance between quaternions p and q.
rowan.geometry.riemann_exp_map	Compute the exponential map on the Riemannian manifold $\mathbb{H}^*$.
rowan.geometry.riemann_log_map	Compute the log map on the Riemannian manifold $\mathbb{H}^*$.
rowan.geometry.intrinsic_distance	Compute the intrinsic distance between quaternions.
rowan.geometry.sym_intrinsic_distance	Compute the symmetrized intrinsic distance between quaternions.
rowan.geometry.angle	Compute the angle of rotation of a quaternion.

Details

Various tools for working with the geometric representation of quaternions.

A particular focus is computing the distance between quaternions. These distance computations can be complicated, particularly good metrics for distance on the Riemannian manifold representing quaternions do not necessarily coin-

cide with good metrics for similarities between rotations. An overview of distance measurements can be found in this paper.

rowan.geometry.**distance**(p, q)

Determine the distance between quaternions p and q.

This is the most basic distance that can be defined on the space of quaternions; it is the metric induced by the norm on this vector space $\rho(p, q) = ||p - q||$.

When applied to unit quaternions, this function produces values in the range $[0, 2]$.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.

Returns Distances between p and q.

Return type (...) numpy.ndarray

Example:

```
>>> rowan.geometry.distance([1, 0, 0, 0], [1, 0, 0, 0])
0.0
```

rowan.geometry.**sym_distance**(p, q)

Determine the distance between quaternions p and q.

This is a symmetrized version of *distance()* that accounts for the fact that p and $-p$ represent identical rotations. This makes it a useful measure of rotation similarity.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.

When applied to unit quaternions, this function produces values in the range $[0, \sqrt{2}]$.

Returns Symmetrized distances between p and q.

Return type (...) numpy.ndarray

Example:

```
>>> rowan.geometry.sym_distance([1, 0, 0, 0], [-1, 0, 0, 0])
0.0
```

rowan.geometry.**riemann_exp_map**(p, v)

Compute the exponential map on the Riemannian manifold $\mathbb{H}^*$.

The nonzero quaternions form a Lie algebra $\mathbb{H}^*$ that is also a Riemannian manifold. In general, given a point p on a Riemannian manifold $\mathcal{M}$ and an element of the tangent space at p, $v \in T_p\mathcal{M}$, the Riemannian exponential map is defined by the geodesic starting at p and tracing out an arc of length v in the direction of v. This function computes the endpoint of that path (which is itself a quaternion).

Explicitly, we define the exponential map as

$$\mathrm{Exp}_p(v) = p\exp(v) \quad (3.12)$$

Parameters

- **p** ((..., 4) numpy.ndarray) – Points on the manifold of quaternions.
- **v** ((..., 4) numpy.ndarray) – Tangent vectors to traverse.

Returns The endpoints of the geodesic starting from p and traveling a distance $||v||$ in the direction of v.

Return type (..., 4) numpy.ndarray

Example:

```
rowan.geometry.riemann_exp_map([1, 0, 0, 0], [-1, 0, 0, 0])
```

rowan.geometry.**riemann_log_map**(p, q)

Compute the log map on the Riemannian manifold $\mathbb{H}^*$.

This function inverts *riemann_exp_map()*. See that function for more details. In brief, given two quaternions p and q, this method returns a third quaternion parameterizing the geodesic passing from p to q. It is therefore an important measure of the distance between the two input quaternions.

Parameters

- **p** ((..., 4) numpy.ndarray) – Starting points (quaternions).
- **q** ((..., 4) numpy.ndarray) – Endpoints (quaternions).

Returns The quaternions pointing from p to q with magnitudes equal to the length of the geodesics joining these quaternions.

Return type (..., 4) numpy.ndarray

Example:

```
>>> rowan.geometry.riemann_log_map([1, 0, 0, 0], [-1, 0, 0, 0])
array([0., 0., 0., 0.])
```

rowan.geometry.**intrinsic_distance**(p, q)

Compute the intrinsic distance between quaternions.

The quaternion distance is determined as the length of the quaternion joining the two quaternions (see *riemann_log_map()*). Rather than computing this directly, however, as shown in [Huynh09] we can compute this distance using the following equivalence:

$$||\log(pq^{-1})|| = 2\cos(|\langle p, q\rangle|) \quad (3.13)$$

When applied to unit quaternions, this function produces values in the range $[0, \pi]$.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.

Returns Intrinsic distances between p and q.

Return type (...) numpy.ndarray

Example:

```
rowan.geometry.intrinsic_distance([1, 0, 0, 0], [-1, 0, 0, 0])
```

rowan.geometry.**sym_intrinsic_distance**(*p*, *q*)

Compute the symmetrized intrinsic distance between quaternions.

This is a symmetrized version of *intrinsic_distance()* that accounts for the double cover $SU(2) \to SO(3)$, making it a more useful metric for rotation similarity.

When applied to unit quaternions, this function produces values in the range $[0, \frac{\pi}{2}]$.

Parameters

- **p** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q** ((..., 4) numpy.ndarray) – Second array of quaternions.

Returns Symmetrized intrinsic distances between p and q.

Return type (...) numpy.ndarray

Example:

```
>>> rowan.geometry.sym_intrinsic_distance([1, 0, 0, 0], [-1, 0, 0, 0])
array(0.)
```

rowan.geometry.**angle**(*p*)

Compute the angle of rotation of a quaternion.

Note that this is identical to `2*intrinsic_distance(p, np.array([1, 0, 0, 0]))`.

Parameters **p** ((..., 4) numpy.ndarray) – Array of quaternions.

Returns Angles traced out by the rotations p.

Return type (...) numpy.ndarray

Example:

```
>>> rowan.geometry.angle([1, 0, 0, 0])
0.0
```

3.4 interpolate

Overview

rowan.interpolate.slerp	Spherical linear interpolation between p and q.
rowan.interpolate.slerp_prime	Compute the derivative of slerp.
rowan.interpolate.squad	Cubically interpolate between p and q.

Details

Interpolate between pairs of quaternions.

The rowan package provides a simple interface to slerp, the standard method of quaternion interpolation for two quaternions.

rowan.interpolate.**slerp**(*q0*, *q1*, *t*, *ensure_shortest=True*)

Spherical linear interpolation between p and q.

The slerp formula can be easily expressed in terms of the quaternion exponential (see *rowan.exp()*).

Parameters

- **q0** ((..., 4) numpy.ndarray) – First array of quaternions.
- **q1** ((..., 4) numpy.ndarray) – Second array of quaternions.
- **t** ((...) numpy.ndarray) – Interpolation parameter $\in [0, 1]$
- **ensure_shortest** (*bool*) – Flip quaternions to ensure we traverse the geodesic in the shorter ($< 180°$) direction.

Note: Given inputs such that $t \notin [0, 1]$, the values outside the range are simply assumed to be 0 or 1 (depending on which side of the interval they fall on).

Returns Interpolations between p and q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> import numpy as np
>>> rowan.interpolate.slerp(
...     [[1, 0, 0, 0]], [[np.sqrt(2)/2, np.sqrt(2)/2, 0, 0]], 0.5)
array([[0.92387953, 0.38268343, 0.        , 0.        ]])
```

rowan.interpolate.**slerp_prime**(*q0*, *q1*, *t*, *ensure_shortest=True*)

Compute the derivative of slerp.

Parameters

- **q0** ((..., 4) numpy.ndarray) – First set of quaternions.
- **q1** ((..., 4) numpy.ndarray) – Second set of quaternions.
- **t** ((...) numpy.ndarray) – Interpolation parameter $\in [0, 1]$
- **ensure_shortest** (*bool*) – Flip quaternions to ensure we traverse the geodesic in the shorter ($< 180°$) direction

Returns The derivative of the interpolations between p and q.

Return type (..., 4) numpy.ndarray

Example:

```
import numpy as np
q_slerp_prime rowan.interpolate.slerp_prime(
    [[1, 0, 0, 0]], [[np.sqrt(2)/2, np.sqrt(2)/2, 0, 0]], 0.5)
```

rowan.interpolate.**squad**(*p*, *a*, *b*, *q*, *t*)

Cubically interpolate between p and q.

The SQUAD formula is just a repeated application of Slerp between multiple quaternions as originally derived in [Shoemake85]:

$$\text{squad}(p, a, b, q, t) = \text{slerp}(p, q, t)\left(\text{slerp}(p, q, t)^{-1}\text{slerp}(a, b, t)\right)^{2t(1-t)} \quad (3.14)$$

Parameters

- **p** ((..., 4) numpy.ndarray) – First endpoint of interpolation.
- **a** ((..., 4) numpy.ndarray) – First control point of interpolation.
- **b** ((..., 4) numpy.ndarray) – Second control point of interpolation.
- **q** ((..., 4) numpy.ndarray) – Second endpoint of interpolation.
- **t** ((...) numpy.ndarray) – Interpolation parameter $t \in [0, 1]$.

Returns Interpolations between p and q.

Return type (..., 4) numpy.ndarray

Example:

```
>>> import numpy as np
>>> rowan.interpolate.squad(
...     [1, 0, 0, 0], [np.sqrt(2)/2, np.sqrt(2)/2, 0, 0],
...     [0, np.sqrt(2)/2, np.sqrt(2)/2, 0],
...     [0, 0, np.sqrt(2)/2, np.sqrt(2)/2], 0.5)
array([[0.64550177, 0.47254009, 0.52564058, 0.28937053]])
```

3.5 mapping

Overview

rowan.mapping.kabsch	Find the optimal rotation and translation to map between two sets of points.
rowan.mapping.davenport	Find the optimal rotation and translation to map between two sets of points.
rowan.mapping.procrustes	Solve the orthogonal Procrustes problem with algorithmic options.
rowan.mapping.icp	Find best mapping using the Iterative Closest Point algorithm.

Details

Solve Procrustes-type problems and perform basic point-cloud registration.

The general space of problems that this subpackage addresses is a small subset of the broader space of point set registration, which attempts to optimally align two sets of points. In general, this mapping can be nonlinear. The restriction of this superposition to linear transformations composed of translation, rotation, and scaling is the study of Procrustes superposition, the first step in the field of Procrustes analysis, which performs the superposition in order to compare two (or more) shapes.

If points in the two sets have a known correspondence, the problem is much simpler. Various precise formulations exist that admit analytical formulations, such as the orthogonal Procrustes problem searching for an orthogonal transformation

$$R = \text{argmin}_{\Omega} ||\Omega A - B||_F, \ \Omega^T \Omega = \mathbb{1} \quad (3.15)$$

or, if a pure rotation is desired, Wahba's problem

$$\min_{\boldsymbol{R}\in SO(3)} \frac{1}{2}\sum_{k=1}^{N} a_k ||\boldsymbol{w}_k - \boldsymbol{R}\boldsymbol{v}_k||^2 \quad (3.16)$$

Numerous algorithms to solve this problem exist, particularly in the field of aerospace engineering and robotics where this problem must be solved on embedded systems with limited processing. Since that constraint does not apply here, this package simply implements some of the most stable known methods irrespective of cost. In particular, this package contains the Kabsch algorithm, which solves Wahba's problem using an SVD in the vein of Peter Schonemann's original solution to the orthogonal Procrustes problem. Additionally this package contains the Davenport q method, which works directly with quaternions. The most popular algorithms for Wahba's problem are variants of the q method that are faster at the cost of some stability; we omit these here.

In addition, *rowan.mapping* also includes some functionality for more general point set registration. If a point cloud has a set of known symmetries, these can be tested explicitly by *rowan.mapping* to find the smallest rotation required for optimal mapping. If no such correspondence is known at all, then the iterative closest point algorithm can be used to approximate the mapping.

rowan.mapping.**kabsch**(*X*, *Y*, *require_rotation=True*)

Find the optimal rotation and translation to map between two sets of points.

This function implements the Kabsch algorithm, which minimizes the RMSD between two sets of points. One benefit of this approach is that the SVD works in dimensions > 3.

Parameters

- **X** ((N, m) numpy.ndarray) – First set of N points.
- **Y** ((N, m) numpy.ndarray) – Second set of N points.
- **require_rotation** (*bool*) – If false, the returned quaternion.

Returns The rotation matrix and translation vector to map X onto Y.

Return type tuple[(m, m) numpy.ndarray, (m,) numpy.ndarray]

Example:

```
>>> import numpy as np

>>> # Create some random points, then make a random transformation of
>>> # these points
>>> points = np.random.rand(10, 3)
>>> rotation = rowan.random.rand(1)
>>> translation = np.random.rand(1, 3)
>>> transformed_points = rowan.rotate(rotation, points) + translation

>>> # Recover the rotation and check
>>> R, t = rowan.mapping.kabsch(points, transformed_points)
>>> q = rowan.from_matrix(R)
```

(continues on next page)

(continued from previous page)

```
>>> assert np.logical_or(
...     np.allclose(rotation, q), np.allclose(rotation, -q))
>>> assert np.allclose(translation, t)
```

rowan.mapping.**davenport**(*X*, *Y*)

Find the optimal rotation and translation to map between two sets of points.

This function implements the Davenport q-method, the most robust method and basis of most modern solvers. It involves the construction of a particular matrix, the Davenport K-matrix, which is then diagonalized to find the appropriate eigenvalues. More modern algorithms aim to solve the characteristic equation directly rather than diagonalizing, which can provide speed benefits at the potential cost of robustness. The implementation in `rowan` does not do this, instead simply computing the spectral decomposition.

Parameters

- **X** ((N, 3) numpy.ndarray) – First set of N points.
- **Y** ((N, 3) numpy.ndarray) – Second set of N points.

Returns The quaternion and translation vector to map `X` onto `Y`.

Return type tuple[(4,) numpy.ndarray, (m,) numpy.ndarray]

Example:

```
>>> import numpy as np

>>> # Create some random points, then make a random transformation of
>>> # these points
>>> points = np.random.rand(10, 3)
>>> rotation = rowan.random.rand(1)
>>> translation = np.random.rand(1, 3)
>>> transformed_points = rowan.rotate(rotation, points) + translation

>>> # Recover the rotation and check
>>> q, t = rowan.mapping.davenport(points, transformed_points)

>>> assert np.logical_or(
...     np.allclose(rotation, q), np.allclose(rotation, -q))
>>> assert np.allclose(translation, t)
```

rowan.mapping.**procrustes**(*X*, *Y*, *method='best'*, *equivalent_quaternions=None*)

Solve the orthogonal Procrustes problem with algorithmic options.

Parameters

- **X** ((N, m) numpy.ndarray) – First set of N points.
- **Y** ((N, m) numpy.ndarray) – Second set of N points.
- **method** (*str*) – A method to use. Options are 'kabsch', 'davenport' and 'horn'. The default is to select the best option ('best').
- **equivalent_quaternions** (*array-like*) – If the precise correspondence is not known, but the points are known to be part of a body with specific symmetries, the set of quaternions generating symmetry-equivalent configurations can be provided. These quaternions will be tested exhaustively to find the smallest symmetry-equivalent rotation.

Returns The quaternion and translation vector to map `X` onto `Y`.

Return type tuple[(4,) numpy.ndarray, (m,) numpy.ndarray]

Example:

```
>>> import numpy as np

>>> # Create some random points, then make a random transformation of
>>> # these points
>>> points = np.random.rand(10, 3)
>>> rotation = rowan.random.rand(1)
>>> translation = np.random.rand(1, 3)
>>> transformed_points = rowan.rotate(rotation, points) + translation

>>> # Recover the rotation and check
>>> q, t = rowan.mapping.procrustes(
...     points, transformed_points, method='horn')

>>> assert np.logical_or(
...     np.allclose(rotation, q), np.allclose(rotation, -q))
>>> assert np.allclose(translation, t)
```

rowan.mapping.**icp**(*X*, *Y*, *method='best'*, *unique_match=True*, *max_iterations=20*, *tolerance=0.001*, *return_indices=False*)

Find best mapping using the Iterative Closest Point algorithm.

Parameters

- **X** ((N, m) numpy.ndarray) – First set of N points.
- **Y** ((N, m) numpy.ndarray) – Second set of N points.
- **method** (*str*) – A method to use for each alignment. Options are 'kabsch', 'davenport' and 'horn'. The default is to select the best option ('best').
- **unique_match** (*bool*) – Whether to require nearest neighbors to be unique.
- **max_iterations** (*int*) – Number of iterations to attempt.
- **tolerance** (*float*) – Indicates convergence.
- **return_indices** (*bool*) – Whether to return indices.

Returns The quaternion and translation vector to map X onto Y. The (optional) last return value is an array of indices mapping points in X to points in Y.

Return type tuple[(4,) numpy.ndarray, (m,) numpy.ndarray`[, (N, ) :class:`numpy.ndarray]]

Example:

```
>>> import numpy as np

>>> # Create some random points
>>> points = np.random.rand(10, 3)

>>> # Only works for small rotations
>>> rotation = rowan.from_axis_angle((1, 0, 0), 0.01)

>>> # Apply a random translation and permutation
>>> translation = np.random.rand(1, 3)
>>> permutation = np.random.permutation(10)
>>> transformed_points = rowan.rotate(
...     rotation, points[permutation]) + translation
```

(continues on next page)

(continued from previous page)

```
>>> # Recover the rotation and check
>>> R, t, indices = rowan.mapping.icp(points, transformed_points,
...                                   return_indices=True)
>>> q = rowan.from_matrix(R)

>>> assert np.logical_or(
...     np.allclose(rotation, q), np.allclose(rotation, -q))
>>> assert np.allclose(translation, t)
>>> assert np.array_equal(permutation, indices)
```

3.6 random

Overview

rowan.random.rand	Generate random rotations uniformly distributed on a unit sphere.
rowan.random.random_sample	Generate random rotations uniformly.

Details

Various functions for generating random sets of rotation quaternions.

Random quaternions in general can be generated by simply randomly sampling 4-vectors, and they can be converted into rotation quaternions by normalizing them. This package is strictly focused on generating uniform samples on $SO(3)$, ensuring that rotations are uniformly sampled rather than just the space of unit quaternions.

rowan.random.**rand**(**args*)

Generate random rotations uniformly distributed on a unit sphere.

This is a convenience function *a la* `np.random.rand`. If you want a function that takes a tuple as input, use *random_sample()* instead.

Parameters **shape** (*tuple*) – The shape of the array to generate.

Returns Random quaternions of the shape provided with an additional axis of length 4.

Return type numpy.ndarray

Example:

```
>>> rowan.random.rand(3, 3, 2)
```

rowan.random.**random_sample**(*size=None*)

Generate random rotations uniformly.

In general, sampling from the space of all quaternions will not generate uniform rotations. What we want is a distribution that accounts for the density of rotations, *i.e.*, a distribution that is uniform with respect to the appropriate measure. The algorithm used here is detailed in [Shoe92].

Parameters **size** (*tuple*) – The shape of the array to generate.

Returns Random quaternions of the shape provided with an additional axis of length 4.

Return type numpy.ndarray

Example:

```
>>> rowan.random.random_sample((3, 3, 2))
```

3.7 Development Guide

All contributions to **rowan** are welcome! Developers are invited to contribute to the framework by pull request to the package repository on github, and all users are welcome to provide contributions in the form of **user feedback** and **bug reports**. We recommend discussing new features in form of a proposal on the issue tracker for the appropriate project prior to development.

3.7.1 Design Philosophy and Code Guidelines

The goal of **rowan** is to provide a flexible, easy-to-use, and scalable approach to dealing with rotation representations. To best serve these goals, **rowan** operates entirely on NumPy arrays (the *de facto* standard for efficient multi-dimensional arrays in Python) and supports NumPy broadcasting wherever possible. Use of broadcasting ensures that **rowan** can take full advantage of NumPy performance, and in general all operations are very fast (benchmarks are included in the code base). Furthermore, to remain lightweight and easy to install, **rowan** is written in **pure Python** and has no hard dependencies aside from NumPy.

Code contributions should keep these ideals in mind and adhere to the following guidelines:

- Use the OneFlow model of development: - Both new features and bug fixes should be developed in branches based on `master`. - Hotfixes (critical bugs that need to be released *fast*) should be developed in a branch based on the latest tagged release.
- All code must be compatible with all supported versions of Python (listed in the package `setup.py` file).
- Avoid external dependencies where possible, and avoid introducing **any** hard dependencies. Soft dependencies are allowed for specific functionality, but such dependencies cannot impede the installation of **rowan** or the use of any other features.
- All code should adhere to the source code and documentation conventions discussed below.
- Create unit tests and integration tests as part of development.
- Preserve backwards-compatibility whenever possible. Make clear if something must change, and notify package maintainers that merging such changes will require a major release.
- Enable broadcasting if at all possible. Functions for which broadcasting is not available must be documented as such.

To provide a reasonable balance between a high level of backwards compatibility and a reasonable maintenance burden, **rowan** has adopted NEP 29 to limit the Python and NumPy versions that will be supported.

Tip: During continuous integration, the code is checked automatically with Flake8. To ensure your code is compliant before committing, you can set up a pre-commit hook using pre-commit (recommended) or use flake8's built-in hooks installation:

```
flake8 --install-hook git
git config --bool flake8.strict true
```

Note: Please see the individual package documentation for detailed guidelines on how to contribute to a specific package.

Source Code Conventions

The **rowan** package adheres to a relatively strict set of style guidelines. All code in **rowan** should be formatted using black; a notable consequence of this is that the recommended max line length is 88, not the more common 80. Imports should be formatted using isort. For guidance on the style, see PEP 8 and the Google Python Style Guide, but any ambiguities should be resolved automatically by running black. All code should also follow the principles in PEP 20. In particular, always prefer simple, explicit code where possible, avoiding unnecessary convolution or complicated code that could be written more simply. Avoid writing code in a manner that will be difficult for others to understand.

Tip: Developers should format their code using black and isort locally using the commands:

```
black --exclude "coxeter/[polytri|bentley_ottman]" coxeter/ tests/
isort -rc coxeter/ tests/
```

Documentation

API documentation should be written as part of the docstrings of the package in the Google style. There is one notable exception to the guide: class properties should be documented in the getters functions, not as class attributes, to allow for more useful help messages and inheritance of docstrings. Docstrings may be validated using pydocstyle (or using the flake8-docstrings plugin as documented above). The official documentation is generated from the docstrings using Sphinx.

Unit Tests

All code should include a set of unit tests which test for correct behavior. All tests should be placed in the `tests` folder at the root of the project. These tests should be as simple as possible, testing a single function each, and they should be kept as short as possible. Tests should also be entirely deterministic: if you are using a random set of objects for testing, they should either be generated once and then stored in the `tests/files` folder, or the random number generator in use should be seeded explicitly (*e.g*, `numpy.random.seed` or `random.seed`). Tests should be written in the style of the standard Python unittest framework. At all times, tests should be executable by simply running `python -m unittest discover tests` from the root of the project.

3.7.2 Release Guide

To make a new release of rowan, follow the following steps:

1. Make a new branch off of master based on the expected new version, *e.g.* release-2.3.1.
2. Make any final changes as desired on this branch. Push the changes and ensure all tests are passing as expected on the new branch.
3. Once the branch is completely finalized, run bumpversion with the appropriate type (patch, minor, major) so that the version now matches the version number in the branch name.
4. Merge the branch back into master, then push master and push tags. The tagged commit will automatically trigger generation of binaries and upload to PyPI and conda-forge.
5. Delete the release branch both locally and on the remote.

3.8 License


```
rowan BSD-3 Clause Open Source Software License

Copyright 2017-2020 The Regents of the University of Michigan
All rights reserved.

rowan may contain modifications ("Contributions") provided, and to which
copyright is held, by various Contributors who have granted The Regents of the
University of Michigan the right to modify and/or distribute such Contributions.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software without
   specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```


3.9 Changelog

The format is based on Keep a Changelog. This project adheres to Semantic Versioning.

3.9.1 v1.3.0 - 2020-06-18

Added

- Extensive new validation using numerous flake8 plugins (including black for code style and pydocstyle for docstrings, among others).

Changed

- Drop support for all Python versions earlier than 3.6 and all NumPy versions before 1.15.

Fixed

- Docstring of geometry.angle was missing a factor of 2 in the comparison to intrinsic_distance.
- Docstrings of functions using support1d decorator were losing their docstring (fixed with functools.wraps).
- Docstrings of return types of all functions.

3.9.2 v1.2.2 - 2019-09-11

Added

- Mapping indices can be returned upon request from mapping.icp.

Fixed

- Euler angle calculations when cos(beta) = 0.

3.9.3 v1.2.1 - 2019-05-30

Added

- Official ContributorAgreement.

Fixed

- Broadcasting for nD arrays of quaternions in to_axis_angle is fixed.
- Providing equivalent quaternions to mapping.procrustes properly performs rotations.

3.9.4 v1.2.0 - 2019-02-12

Changed

- Code is now hosted on GitHub.

Fixed

- Various style issues.

3.9.5 v1.1.7 - 2019-01-23

Changed

- Stop requiring unit quaternions for rotation and reflection (allows scaling).

3.9.6 v1.1.6 - 2018-10-18

Fixed

- Fifth try of releasing using CircleCI.

3.9.7 v1.1.5 - 2018-10-18

Fixed

- Fourth try of releasing using CircleCI.

3.9.8 v1.1.4 - 2018-10-18

Fixed

- Third try of releasing using CircleCI.

3.9.9 v1.1.3 - 2018-10-18

Fixed

- Second try of releasing using CircleCI.

3.9.10 v1.1.2 - 2018-10-18

Fixed

- Fix usage of release tag in CircleCI config.

3.9.11 v1.1.1 - 2018-10-18

Added

- Automated deployment using CircleCI.
- Added PDF of paper to the repository.

Fixed

- Added missing factor of 2 in angle calculation.
- Fixed issue where method was not respected in rowan.mapping.
- Disabled equivalent quaternion feature and test of rowan.mapping, which has a known bug.
- Added missing negative in failing unit test.

3.9.12 v1.1.0 - 2018-07-30

Added

- Included benchmarks including comparison to alternatives.
- Installation instructions in the Sphinx documentation.
- More examples for rowan.mapping.

Changed

- All examples in docstrings now use the full paths of subpackages.
- All examples in docstrings import all needed packages aside from rowan.

Fixed

- Instability in vector_vector_rotation for antiparallel vectors.
- Various code style issues.
- Broken example in the Sphinx documentation.

3.9.13 v1.0.0 - 2018-05-29

Fixed

- Numerous style fixes.
- Fix version numbering in the Changelog.

3.9.14 v0.6.1 - 2018-04-20

Fixed

- Use of bumpversion and consistent versioning across the package.

3.9.15 v0.6.0 - 2018-04-20

Added

- Derivatives and integrals of quaternions.
- Point set registration methods and Procrustes analysis.

3.9.16 v0.5.1 - 2018-04-13

Fixed

- README rendering on PyPI.

3.9.17 v0.5.0 - 2018-04-12

Added

- Various distance metrics on quaternion space.
- Quaternion interpolation.

Fixed

- Update empty __all__ variable in geometry to export functions.

3.9.18 v0.4.4 - 2018-04-10

Added

- Rewrote internals for upload to PyPI.

3.9.19 v0.4.3 - 2018-04-10

Fixed

- Typos in documentation.

3.9.20 v0.4.2 - 2018-04-09

Added

- Support for Read The Docs and Codecov.
- Simplify CircleCI testing suite.
- Minor changes to README.
- Properly update this document.

3.9.21 v0.4.1 - 2018-04-08

Fixed

- Exponential for bases other than e are calculated correctly.

3.9.22 v0.4.0 - 2018-04-08

Added

- Add functions relating to exponentiation: exp, expb, exp10, log, logb, log10, power.
- Add core comparison functions for equality, closeness, finiteness.

3.9.23 v0.3.0 - 2018-03-31

Added

- Broadcasting works for all methods.
- Quaternion reflections.
- Random quaternion generation.

Changed

- Converting from Euler now takes alpha, beta, and gamma as separate args.
- Ensure more complete coverage.

3.9.24 v0.2.0 - 2018-03-08

Added

- Added documentation.
- Add tox support.
- Add support for range of python and numpy versions.
- Add coverage support.

Changed

- Clean up CI.
- Ensure pep8 compliance.

3.9.25 v0.1.0 - 2018-02-26

Added

- Initial implementation of all functions.

3.10 Credits

The following people contributed to the *rowan* package.

Vyas Ramasubramani <vramasub@umich.edu>, University of Michigan - **Lead developer**.

- Initial design.
- Wrote quaternion operations.
- Wrote calculus subpackage.
- Wrote geometry subpackage.
- Wrote interpolate subpackage.

- Wrote mapping subpackage.
- Wrote random subpackage.
- Wrote documentation.

Bradley Dice <bdice@bradleydice.com>, University of Michigan

- Code review.
- JOSS paper review.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[Itzhack00] Itzhack Y. Bar-Itzhack. “New Method for Extracting the Quaternion from a Rotation Matrix”, Journal of Guidance, Control, and Dynamics, Vol. 23, No. 6 (2000), pp. 1085-1087 https://doi.org/10.2514/2.4654

[Huynh09] Huynh DQ (2009) Metrics for 3D rotations: comparison and analysis. J Math Imaging Vis 35(2):155-164

[Shoemake85] Ken Shoemake. Animating rotation with quaternion curves. SIGGRAPH Comput. Graph., 19(3):245-254, July 1985.

[Shoe92] Shoemake, K.: Uniform random rotations. In: D. Kirk, editor, Graphics Gems III, pages 124-132. Academic, New York, 1992.

Python Module Index

r

rowan, 10
rowan.calculus, 20
rowan.geometry, 21
rowan.interpolate, 24
rowan.mapping, 26
rowan.random, 30

Index

A

allclose() (*in module rowan*), 10
angle() (*in module rowan.geometry*), 24

C

conjugate() (*in module rowan*), 10

D

davenport() (*in module rowan.mapping*), 28
derivative() (*in module rowan.calculus*), 20
distance() (*in module rowan.geometry*), 22
divide() (*in module rowan*), 10

E

equal() (*in module rowan*), 12
exp() (*in module rowan*), 11
exp10() (*in module rowan*), 11
expb() (*in module rowan*), 11

F

from_axis_angle() (*in module rowan*), 12
from_euler() (*in module rowan*), 12
from_matrix() (*in module rowan*), 13
from_mirror_plane() (*in module rowan*), 13

I

icp() (*in module rowan.mapping*), 29
integrate() (*in module rowan.calculus*), 21
intrinsic_distance() (*in module rowan.geometry*), 23
inverse() (*in module rowan*), 13
is_unit() (*in module rowan*), 15
isclose() (*in module rowan*), 14
isfinite() (*in module rowan*), 14
isinf() (*in module rowan*), 14
isnan() (*in module rowan*), 14

K

kabsch() (*in module rowan.mapping*), 27

L

log() (*in module rowan*), 15
log10() (*in module rowan*), 16
logb() (*in module rowan*), 15

M

multiply() (*in module rowan*), 16

N

norm() (*in module rowan*), 16
normalize() (*in module rowan*), 17
not_equal() (*in module rowan*), 17

P

power() (*in module rowan*), 17
procrustes() (*in module rowan.mapping*), 28

R

rand() (*in module rowan.random*), 30
random_sample() (*in module rowan.random*), 30
reflect() (*in module rowan*), 17
riemann_exp_map() (*in module rowan.geometry*), 22
riemann_log_map() (*in module rowan.geometry*), 23
rotate() (*in module rowan*), 18
rowan (*module*), 10
rowan.calculus (*module*), 20
rowan.geometry (*module*), 21
rowan.interpolate (*module*), 24
rowan.mapping (*module*), 26
rowan.random (*module*), 30

S

slerp() (*in module rowan.interpolate*), 24
slerp_prime() (*in module rowan.interpolate*), 25
squad() (*in module rowan.interpolate*), 25
sym_distance() (*in module rowan.geometry*), 22

sym_intrinsic_distance() (*in module rowan.geometry*), 24

T

to_axis_angle() (*in module rowan*), 18
to_euler() (*in module rowan*), 18
to_matrix() (*in module rowan*), 19

V

vector_vector_rotation() (*in module rowan*), 20