rowan Documentation

Release 1.2.1

Vyas Ramasubramani

May 30, 2019

Modules:

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.

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

rowan

Overview

`rowan.conjugate`	Conjugates an array of quaternions.
`rowan.inverse`	Computes the inverse of an array of quaternions.
`rowan.exp`	Computes the natural exponential function e^q.
`rowan.expb`	Computes the exponential function b^q.
`rowan.exp10`	Computes the exponential function 10^q.
`rowan.log`	Computes the quaternion natural logarithm.
`rowan.logb`	Computes the quaternion logarithm to some base b.
`rowan.log10`	Computes 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.
`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.

Continued on next page

Table 1 – continued from previous page

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

Returns Boolean indicating whether or not all quaternions are close.

Example:

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

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

Conjugates an array of quaternions.

Parameters **q** (*(.., 4) np.array*) – Array of quaternions.

Returns Array of shape (. . .) containing conjugates of q.

Example:

```
q_star = rowan.conjugate([1, 0, 0, 0])
```

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

Divides two arrays of quaternions.

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

Parameters

- **qi** (*(.., 4) np.array*) – Dividend quaternions.
- **qj** (*(.., 4) np.array*) – Divisor quaternions.

Returns Array of shape (. . .) containing element-wise quotients of qi and qj.

Example:

```
quot = rowan.divide([1, 0, 0, 0], [2, 0, 0, 0])
```

`rowan.`**`exp`**`(`*q*`)`

Computes 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:

$$e^q = e^{a+v} \tag{1.1}$$

$$= e^a \left(\sum_{k=0}^{\infty} \frac{v^k}{k!} \right) \tag{1.2}$$

$$= e^a \left(\cos||\boldsymbol{v}|| + \frac{\boldsymbol{v}}{||\boldsymbol{v}||} \sin||\boldsymbol{v}|| \right) \tag{1.3}$$

Parameters **q** (*(.., 4) np.array*) – Array of quaternions.

Returns Array of shape (...) containing exponentials of q.

Example:

```
q_exp = rowan.exp([1, 0, 0, 0])
```

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

Computes 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:

$$b^q = y \tag{1.4}$$

$$q = \log_b y = \frac{\ln y}{\ln b} \tag{1.5}$$

$$y = e^{q \ln b} \tag{1.6}$$

Parameters **q** (*(.., 4) np.array*) – Array of quaternions.

Returns Array of shape (...) containing exponentials of q.

Example:

```
q_exp = rowan.expb([1, 0, 0, 0], 2)
```

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

Computes the exponential function 10^q.

Wrapper around `expb()`.

Parameters **q** (*(.., 4) np.array*) – Array of quaternions.

Returns Array of shape (...) containing exponentials of q.

Example:

```
q_exp = rowan.exp10([1, 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) np.array*) – First array of quaternions.

- **q** (*(..,4) np.array*) – Second array of quaternions.

Returns A boolean array of shape (. . .) indicating equality.

Example:

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

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

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

Parameters

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

Returns Array of shape (. . . , 4) containing the corresponding rotation quaternions.

Example:

```
quat = rowan.from_axis_angle([[1, 0, 0]], np.pi/3)
```

`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** (*(..) np.array*) – Array of α values in radians.
- **beta** (*(..) np.array*) – Array of β values in radians.
- **gamma** (*(..) np.array*) – 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 Array of shape (. . . , 4) containing quaternions corresponding to the input angles.

Example:

```
q1 = rowan.from_euler(0.3, 0.5, 0.7)
```

`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) np.array*) – An array of rotation matrices.

Returns Array of shape (. . . , 4) containing the corresponding rotation quaternions.

Example:

```
q1 = rowan.from_matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

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** (*(..) np.array*) – First planar component.
- **y** (*(..) np.array*) – Second planar component.
- **z** (*(..) np.array*) – Third planar component.

Returns Array of shape (...) containing quaternions reflecting about the input plane (x, y, z).

Example:

```
quat_ref = rowan.from_mirror_plane(*(1, 2, 3))
```

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

Computes the inverse of an array of quaternions.

Parameters **q** (*(..,4) np.array*) – Array of quaternions.

Returns Array of shape (...) containing inverses of q.

Example:

```
q_inv = rowan.inverse([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) np.array*) – First array of quaternions.
- **q** (*(..,4) np.array*) – Second array of quaternions.
- ****kwargs** – Keyword arguments to pass to np.isclose.

Returns A boolean array of shape (...) indicating which quaternions are close.

Example:

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

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

Test element-wise for infinite quaternions.

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

Parameters **q** (*(..,4) np.array*) – Array of quaternions

Returns A boolean array of shape (...) indicating infinite quaternions.

Example:

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

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

Test element-wise for finite quaternions.

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

Parameters **q** (*(..,4) np.array*) – Array of quaternions.

Returns A boolean array of shape (...) indicating finite quaternions.

Example:

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

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

Test element-wise for NaN quaternions.

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

Parameters **q** (*(..,4) np.array*) – Array of quaternions.

Returns A boolean array of shape (...) indicating whether or not the input quaternions were NaN.

Example:

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

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

Check if all input quaternions have unit norm.

Parameters **q** (*(..,4) np.array*) – Array of quaternions.

Returns Whether or not all inputs are unit quaternions

Return type bool

Example:

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

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

Computes 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:

$$\ln(q) = \ln||q|| + \frac{\boldsymbol{v}}{||\boldsymbol{v}||} \arccos\left(\frac{a}{q}\right) \quad (1.7)$$

Parameters **q** (*(..,4) np.array*) – Array of quaternions.

Returns Array of shape (...) containing logarithms of q.

Example:

```
ln_q  = rowan.log([1, 0, 0, 0])
```

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

Computes 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.

$$\log_b q = y \tag{1.8}$$

$$q = b^y \tag{1.9}$$

$$\ln q = y \ln b \tag{1.10}$$

$$y = \log_b q = \frac{\ln q}{\ln b} \tag{1.11}$$

Parameters

- **`q`** (*`(..,4) np.array`*) – Array of quaternions.
- **`n`** (*`(..) np.array`*) – Scalars to use as log bases.

Returns Array of shape (…) containing logarithms of q.

Example:

```
log2_q = rowan.logb([1, 0, 0, 0], 2)
```

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

Computes the quaternion logarithm base 10.

Wrapper around *`logb()`*.

Parameters **`q`** (*`(..,4) np.array`*) – Array of quaternions.

Returns Array of shape (…) containing logarithms of q.

Example:

```
log10_q = rowan.log10([1, 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) np.array`*) – Array of left quaternions.
- **`qj`** (*`(..,4) np.array`*) – Array of right quaternions.

Returns Array of shape (…) containing element-wise products of q.

Example:

```
prod = rowan.multiply([1, 0, 0, 0], [2, 0, 0, 0])
```

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

Compute the quaternion norm.

Parameters **`q`** (*`(..,4) np.array`*) – Array of quaternions.

Returns Array of shape (…) containing norms of q.

Example:

```
norms = rowan.norm([10, 0, 0, 0])
```

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

Normalize quaternions.

Parameters **q** (*(..,4) np.array*) – Array of quaternions.

Returns Array of shape (. . .) of normalized quaternions.

Example:

```
u = rowan.normalize([10, 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) np.array*) – First array of quaternions.
- **q** (*(..,4) np.array*) – Second array of quaternions.

Returns A boolean array of shape (. . .) indicating inequality.

Example:

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

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

Computes the power of a quaternion q^n.

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

Parameters

- **q** (*(..,4) np.array*) – Array of quaternions.
- **n** (*(..) np.arrray*) – Scalars to exponentiate quaternions with.

Returns Array of shape (. . .) containing powers of q.

Example:

```
q_5 = rowan.power([1, 0, 0, 0], 5)
```

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) np.array*) – Array of quaternions.
- **v** (*(..,3) np.array*) – Array of vectors.

Returns Array of shape (. . . , 3) containing reflections of v.

Example:

```
v_reflected = rowan.reflect([1, 0, 0, 0], [1, 1, 1])
```

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

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

Parameters

- **q** (*(.., 4) np.array*) – Array of quaternions.
- **v** (*(.., 3) np.array*) – Array of vectors.

Returns Array of shape (. . . , 3) containing rotations of v.

Example:

```
v_rot = rowan.reflect([1, 0, 0, 0], [1, 1, 1])
```

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

Convert the quaternions in q to axis angle representations.

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

Returns A tuple of np.arrays (axes, angles) where axes has shape (. . . ,3) and angles has shape (. . . ,1). The angles are in radians.

Example:

```
quat = rowan.to_axis_angle([[1, 0, 0, 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 pi 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:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$$
$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 1 & \cos\theta \end{pmatrix}$$
$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

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).

Parameters

- **q** (*(.., 4) np.array*) – 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 math:*(alpha, beta, gamma)* as the last dimension (in radians).

Return type Array of shape (.., 3) containing Euler angles

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) np.array*) – An array of quaternions.

Returns Array of shape (…, 3, 3) containing the corresponding rotation matrices.

Example:

```
ql = rowan.to_matrix([1, 0, 0, 0]])
```

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

Find the quaternion to rotate one vector onto another.

Parameters

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

Returns Array of shape (…, 4) containing quaternions that rotate v1 onto v2.

Example:

```
q_rot = rowan.vector_vector_rotation([1, 0, 0], [0, 1, 0])
```

CHAPTER 2

calculus

Overview

rowan.calculus.derivative	Compute the instantaneous derivative of unit quaternions, which is defined as
rowan.calculus.integrate	Integrate unit quaternions by angular velocity using the following equation:

Details

This subpackage provides the ability to compute the derivative and integral of a quaternion.

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

Compute the instantaneous derivative of unit quaternions, which is defined as

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

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

Parameters

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

Returns Array of shape $(\ldots, 4)$ containing element-wise derivatives of q.

Example:

```
q_prime = rowan.calculus.derivative([1, 0, 0, 0], [1, 0, 0])
```

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

Integrate unit quaternions by angular velocity using 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) np.array*) – Array of quaternions.
- **v** (*(..,3) np.array*) – Array of angular velocities.
- **dt** (*(..) np.array*) – Array of timesteps.

Returns Array of shape (. . . , 4) containing element-wise integrals of q.

Example:

```
v_next = rowan.calculus.integrate([1, 0, 0, 0], [0, 0, 1e-2], 1)
```

CHAPTER 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 $\mathbb{H}^*$ of nonzero quaterions.
rowan.geometry.riemann_log_map	Compute the log map on the Riemannian manifold $\mathbb{H}^*$ of nonzero quaterions.
rowan.geometry.intrinsic_distance	Compute the intrinsic distance between quaternions on the manifold of quaternions.
rowan.geometry.sym_intrinsic_distance	Compute the intrinsic distance between quaternions on the manifold of quaternions.
rowan.geometry.angle	Compute the angle of rotation of a quaternion.

Details

This subpackage provides 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 coincide 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) np.array*) – First array of quaternions.
- **q** (*(.., 4) np.array*) – Second array of quaternions.

Returns Array of shape (. . .) containing the element-wise distances between the two sets of quaternions.

Example:

```
rowan.geometry.distance([1, 0, 0, 0], [1, 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) np.array*) – First array of quaternions.
- **q** (*(.., 4) np.array*) – Second array of quaternions.

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

Returns Array of shape (. . .) containing the element-wise symmetrized distances between the two sets of quaternions.

Example:

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

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

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

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

$\mathrm{Exp}_p(v) = p\exp(v)(3.1)$

Parameters

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

Returns Array of shape (. . . , 4) containing the endpoints of the geodesic starting from p and traveling a distance $||v||$ in the direction of v.

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 $\mathbb{H}^*$ of nonzero quaterions.

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) np.array*) – Starting points (quaternions).
- **q** (*(.., 4) np.array*) – Endpoints (quaternions).

Returns Array of shape $(\ldots, 4)$ containing quaternions pointing from p to q with magnitudes equal to the length of the geodesics joining these quaternions.

Example:

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

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

Compute the intrinsic distance between quaternions on the manifold of 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:

$$||\log(pq^{-1})|| = 2\cos(|\langle p, q\rangle|)(3.2)$$

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

Parameters

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

Returns Array of shape $(\ldots)$ containing the element-wise intrinsic distances between the two sets of quaternions.

Example:

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

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

Compute the intrinsic distance between quaternions on the manifold of quaternions.

This is a symmetrized version of *intrinsic_distance()* that accounts for the double cover $SU(2) \rightarrow SO(3)$, 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) np.array*) – First array of quaternions.
- **q** (*(.., 4) np.array*) – Second array of quaternions.

Returns Array of shape $(\ldots)$ containing the element-wise symmetrized intrinsic distances between the two sets of quaternions.

Example:

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

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

Compute the angle of rotation of a quaternion.

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

Parameters **`p`** (*`(..,4) np.array`*) – Array of quaternions.

Returns Array of shape (. . .) containing the element-wise angles traced out by these rotations.

Example:

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

CHAPTER 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

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) np.array*) – First array of quaternions.
- **q1** (*(.., 4) np.array*) – Second array of quaternions.
- **t** (*(..) np.array*) – 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 Array of shape $(\ldots, 4)$ containing the element-wise interpolations between p and q.

Example:

```
import numpy as np
q_slerp = rowan.interpolate.slerp(
    [[1, 0, 0, 0]], [[np.sqrt(2)/2, np.sqrt(2)/2, 0, 0]], 0.5)
```

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

Compute the derivative of slerp.

Parameters

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

Returns An array of shape $(\ldots, 4)$ containing the element-wise derivatives of interpolations between p and q.

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]:

$$\text{squad}(p, a, b, q, t) = \text{slerp}(p, q, t)\left(\text{slerp}(p, q, t)^{-1}\text{slerp}(a, b, t)\right)^{2t(1-t)} \quad (4.1)$$

Parameters

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

Returns An array containing the element-wise interpolations between p and q.

Example:

```
import numpy as np
q_squad = 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)
```

CHAPTER 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

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

$$R = \text{argmin}_\Omega ||\Omega A - B||_F,\ \Omega^T\Omega = \mathbb{1} \quad (5.1)$$

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

$$\min_{\boldsymbol{R} \in SO(3)} \frac{1}{2} \sum_{k=1}^{N} a_k ||\boldsymbol{w}_k - \boldsymbol{R}\boldsymbol{v}_k||^2 (5.2)$$

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 knowna 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) np.array*) – First set of N points.
- **Y** (*(N, m) np.array*) – Second set of N points.
- **require_rotation** (*bool*) – If false, the returned quaternion.

Returns A tuple (R, t) where R is the (m x m) rotation matrix to rotate the points and t is the translation.

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)

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) np.array*) – First set of N points.
- **Y** (*(N, 3) np.array*) – Second set of N points.

Returns A tuple (q, t) where q is the quaternion to rotate the points and t is the translation.

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) np.array*) – First set of N points.
- **Y** (*(N, m) np.array*) – 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 A tuple (q, t) where q is the quaternion to rotate the points and t is the translation.

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
```

(continues on next page)

(continued from previous page)

```python

# 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*)

Find best mapping using the Iterative Closest Point algorithm.

Parameters

- **X** (*(N, m) np.array*) – First set of N points.
- **Y** (*(N, m) np.array*) – 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.

Returns A tuple (R, t) where R is the matrix to rotate the points and t is the translation.

Example:

```python
import numpy as np

# Create some random points, then make a random transformation of
# these points
points = np.random.rand(10, 3)

# Only works for small rotations
rotation = rowan.from_axis_angle((1, 0, 0), 0.01)
translation = np.random.rand(1, 3)
transformed_points = rowan.rotate(rotation, points) + translation

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

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

CHAPTER 6

random

Overview

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

Details

Various functions for generating random sets of rotation quaternions. Note that if you simply want random quaternions not restricted to $SO(3)$ you can just generate these directly using `np.random.rand(... 4)`. This subpackage is entirely focused on generating rotation quaternions.

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

Generate random rotations that are 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.

Example:

```
q_rand = 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.

Example:

```
q_rand = rowan.random.random_sample((3, 3, 2))
```

CHAPTER 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.

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 ensure maximum flexibility, **rowan** operates entirely on NumPy arrays, which serve as the *de facto* standard for efficient multi-dimensional arrays in Python. To be available for a wide variety of applications, **rowan** works for arbitrarily shaped NumPy arrays, mimicking NumPy broadcasting to the maximum extent possible. **rowan** is meant to be as lightweight and easy to install as possible. Although it is designed to provide good performance, it is written in **pure Python** and as such may not be the correct choice in cases where the performance of quaternion operations is a critical bottleneck.

All code contributed to **rowan** must 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 conventions discussed below.
- Follow the rules for documentation discussed below.
- Create unit tests and integration tests that cover the common cases and the corner cases of the code (more information below).

- 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.
- For consistency, NumPy should **always** be imported as `np` in code: `import numpy as np`.

Tip: During continuous integration, the code is checked automatically with Flake8. Run the following commands to set up a pre-commit hook that will ensure your code is compliant before committing:

```
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.

7.1.1 Source Code Conventions

All code in rowan should follow PEP 8 guidelines, which are the *de facto* standard for Python code. In addition, follow the Google Python Style Guide, which is largely a superset of PEP 8. Note that Google has amended their standards to match PEP 8's 4 spaces guideline, so write code accordingly.

All code should 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.

7.1.2 Documentation

API documentation should be written as part of the docstrings of the package. All docstrings should be written in the Google style.

Python example:

```python
# This is the correct style
def multiply(x, y):
    """Multiply two numbers

    Args:
        x (float): The first number
        y (float): The second number

    Returns:
        The product
    """

# This is the incorrect style
def multiply(x, y):
    """Multiply two numbers

    :param x: The first number
    :type x: float
```

(continues on next page)

(continued from previous page)

```
    :param y: The second number
    :type y: float
    :returns: The product
    :rtype: float
    """
```

Documentation must be included for all functions in all files. The official documentation is generated from the docstrings using Sphinx.

In addition to API documentation, inline comments are **highly encouraged**. Code should be written as transparently as possible, so the primary goal of documentation should be explaining the algorithms or mathematical concepts underlying the code. Avoid comments that simply restate the nature of lines of code. For example, the comment "compute the spectral decomposition of A" is uninformative, since the code itself should make this obvious, *e.g*, `np.linalg.eigh`. On the other hand, the comment "the eigenvector corresponding to the largest eigenvalue of the A matrix is the quaternion" is instructive.

7.1.3 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.

7.2 Release Guide

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

1. Make a new branch off of develop 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.

CHAPTER 8

License

CHAPTER 9

Changelog

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

9.1 Unreleased

9.1.1 Added

- Official ContributorAgreement.

9.1.2 Fixed

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

9.2 v1.2.0 - 2019-02-12

9.2.1 Changed

- Code is now hosted on GitHub.

9.2.2 Fixed

- Various style issues.

9.3 v1.1.7 - 2019-01-23

9.3.1 Changed

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

9.4 v1.1.6 - 2018-10-18

9.4.1 Fixed

- Fifth try of releasing using CircleCI.

9.5 v1.1.5 - 2018-10-18

9.5.1 Fixed

- Fourth try of releasing using CircleCI.

9.6 v1.1.4 - 2018-10-18

9.6.1 Fixed

- Third try of releasing using CircleCI.

9.7 v1.1.3 - 2018-10-18

9.7.1 Fixed

- Second try of releasing using CircleCI.

9.8 v1.1.2 - 2018-10-18

9.8.1 Fixed

- Fix usage of release tag in CircleCI config.

9.9 v1.1.1 - 2018-10-18

9.9.1 Added

- Automated deployment using CircleCI.

- Added PDF of paper to the repository.

9.9.2 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.

9.10 v1.1.0 - 2018-07-30

9.10.1 Added

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

9.10.2 Changed

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

9.10.3 Fixed

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

9.11 v1.0.0 - 2018-05-29

9.11.1 Fixed

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

9.12 v0.6.1 - 2018-04-20

9.12.1 Fixed

- Use of bumpversion and consistent versioning across the package.

9.13 v0.6.0 - 2018-04-20

9.13.1 Added

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

9.14 v0.5.1 - 2018-04-13

9.14.1 Fixed

- README rendering on PyPI.

9.15 v0.5.0 - 2018-04-12

9.15.1 Added

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

9.15.2 Fixed

- Update empty __all__ variable in geometry to export functions.

9.16 v0.4.4 - 2018-04-10

9.16.1 Added

- Rewrote internals for upload to PyPI.

9.17 v0.4.3 - 2018-04-10

9.17.1 Fixed

- Typos in documentation.

9.18 v0.4.2 - 2018-04-09

9.18.1 Added

- Support for Read The Docs and Codecov.

- Simplify CircleCI testing suite.
- Minor changes to README.
- Properly update this document.

9.19 v0.4.1 - 2018-04-08

9.19.1 Fixed

- Exponential for bases other than e are calculated correctly.

9.20 v0.4.0 - 2018-04-08

9.20.1 Added

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

9.21 v0.3.0 - 2018-03-31

9.21.1 Added

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

9.21.2 Changed

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

9.22 v0.2.0 - 2018-03-08

9.22.1 Added

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

9.22.2 Changed

- Clean up CI.
- Ensure pep8 compliance.

9.23 v0.1.0 - 2018-02-26

9.23.1 Added

- Initial implementation of all functions.

CHAPTER 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 11

Getting Started

11.1 Requirements

The minimum requirements for using rowan are:

- Python = 2.7, >= 3.3
- NumPy >= 1.10

11.2 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
```

11.3 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])
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)
```

11.4 Running Tests

The package is currently tested for Python versions 2.7 and Python >= 3.3 on Unix-like systems. Continuous integrated testing is performed using CircleCI on these Python versions with NumPy versions 1.10 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
```

11.5 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.

11.6 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 12

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 13

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

Index